

**Secure execution environments through  
reconfigurable lightweight cryptographic components**

by

Mahadevan Gomathisankaran

A dissertation submitted to the graduate faculty  
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:  
Akhilesh Tyagi, Major Professor  
Soma Chaudhuri  
Thomas Daniels  
Randall Geiger  
Zhao Zhang

Iowa State University

Ames, Iowa

2006

Copyright © Mahadevan Gomathisankaran, 2006. All rights reserved.

Graduate College  
Iowa State University

This is to certify that the doctoral dissertation of  
Mahadevan Gomathisankaran  
has met the dissertation requirements of Iowa State University

---

Major Professor

---

For the Major Program

## TABLE OF CONTENTS

<b>LIST OF TABLES</b> . . . . .	vi
<b>LIST OF FIGURES</b> . . . . .	vii
<b>CHAPTER 1. OVERVIEW</b> . . . . .	1
<b>CHAPTER 2. ARC3D: ARCHITECTURE SUPPORT FOR 3D OBFUSCATION</b> . . . . .	3
2.1 Introduction . . . . .	3
2.2 The Problem . . . . .	4
2.3 Previous Research . . . . .	7
2.3.1 ABYSS . . . . .	7
2.3.2 TrustNo1 Cryptoprocessor . . . . .	7
2.3.3 XOM . . . . .	8
2.3.4 HIDE . . . . .	8
2.3.5 Oblivious RAM . . . . .	9
2.3.6 Dallas Semiconductor . . . . .	9
2.3.7 Obfuscation . . . . .	9
2.3.8 Analysis . . . . .	9
2.4 Proposed Architecture: Arc3D . . . . .	10
2.4.1 Obfuscation Schema . . . . .	10
2.4.2 Overall Schema . . . . .	14
2.4.3 Reconfigurable Bijective Function Unit . . . . .	15
2.4.4 Obfuscating the Sequence . . . . .	19
2.4.5 Obfuscating the Contents . . . . .	20
2.4.6 Obfuscating Temporal Order (Second-Order Address Sequences) . . . . .	21

2.5	Arc3D in Operation . . . . .	21
2.5.1	Software Distribution . . . . .	22
2.5.2	Management of Protected Process . . . . .	27
2.6	Discussion . . . . .	33
2.6.1	Assumptions . . . . .	33
2.6.2	Attack Scenarios . . . . .	34
2.7	Performance Analysis . . . . .	36
2.8	Conclusion . . . . .	38
<b>CHAPTER 3. TIVA: TRUSTED INTEGRITY VERIFICATION ARCHITECTURE . . . . .</b>		<b>39</b>
3.1	Introduction . . . . .	39
3.2	The Problem . . . . .	41
3.3	The Solution . . . . .	42
3.4	Area and Delay Estimation of RPU . . . . .	47
3.5	Integrity Verification Architecture . . . . .	50
3.5.1	XRPU . . . . .	50
3.5.2	Verification . . . . .	50
3.5.3	Overhead Estimation . . . . .	51
3.6	Discussion . . . . .	52
3.6.1	Obfuscation Strength of Permutation Function (RPU) . . . . .	52
3.6.2	Attack Scenarios . . . . .	55
3.6.3	Flexibility of TIVA . . . . .	55
3.7	Related Work . . . . .	56
3.8	Conclusions . . . . .	57
<b>CHAPTER 4. REBEL: RECONFIGURABLE BLOCK ENCRYPTION LOGIC . . . . .</b>		<b>59</b>
4.1	Introduction . . . . .	59
4.2	Preliminaries . . . . .	60
4.2.1	Notations . . . . .	60
4.2.2	Properties of $G_n^b$ . . . . .	61

4.3	Function Family $F_n^N$ . . . . .	66
4.3.1	Notations . . . . .	66
4.3.2	Construction of $F_n^N$ . . . . .	68
4.3.3	Properties of $F_n^N$ . . . . .	70
4.4	<i>REBEL</i> function family $R_n^{2N}$ . . . . .	79
4.4.1	Construction of $R_n^{2N}$ . . . . .	79
4.4.2	Adversary Models . . . . .	80
4.4.3	Complexity Analysis (PTRA) . . . . .	81
4.4.4	Statistical Adversary Advantage . . . . .	82
4.4.5	Resilience to Cryptanalysis . . . . .	85
4.4.6	Implementation: . . . . .	87
4.5	Conclusion . . . . .	87
<b>CHAPTER 5. CONCLUSION . . . . .</b>		<b>88</b>
<b>REFERENCES . . . . .</b>		<b>90</b>
<b>ACKNOWLEDGEMENTS . . . . .</b>		<b>94</b>

## LIST OF TABLES

Table 2.1	Redundancy Estimation (# of Random Configs = $2^{20}$ ) . . . . .	19
Table 2.2	Memory Hierarchy Simulation Parameters . . . . .	36
Table 2.3	Simulation Results . . . . .	37
Table 3.1	Area Estimate of RPU . . . . .	48
Table 3.2	Delay Estimate of RPU . . . . .	49
Table 3.3	Latency and Area Overhead Estimation of XRPU . . . . .	52
Table 3.4	Average <i>Obfuscation Strength</i> for $2^{20}$ Runs . . . . .	54
Table 4.1	Properties of two instances of $R_n^{2N}$ . . . . .	87

## LIST OF FIGURES

Figure 2.1	Weakness of HIDE approach . . . . .	10
Figure 2.2	Overall Schema of <i>Arc3D</i> Architecture . . . . .	11
Figure 2.3	Static and Dynamic obfuscation . . . . .	12
Figure 2.4	Reconfigurable Bijective Obfuscation Unit . . . . .	15
Figure 2.5	Configuration Selection for each LUT . . . . .	17
Figure 2.6	Three Party Trust Model . . . . .	22
Figure 2.7	Page Obfuscation Function . . . . .	24
Figure 2.8	Extended Binary Format . . . . .	26
Figure 3.1	An Example <i>Hash</i> or <i>Checksum</i> Function $\mathcal{F}$ . . . . .	46
Figure 3.2	A Typical Schema for 5x5-LUT . . . . .	47
Figure 3.3	Integrity Verification Architecture . . . . .	49
Figure 3.4	An Example PPC Micro-Code Implementation of $\mathcal{F}$ . . . . .	51
Figure 3.5	An Example Permutation . . . . .	54
Figure 4.1	Bias Propagation in $n$ -to-1 Gate . . . . .	65
Figure 4.2	Diagrammatic representation of $\Gamma$ . . . . .	67
Figure 4.3	Diagrammatic representation of $F_n^N$ . . . . .	68
Figure 4.4	Construction of $F_n^N$ and <i>Key</i> (Gate Configuration) Assignment . . . . .	69
Figure 4.5	One of the Maximum Controllability Paths in $\Gamma$ . . . . .	72
Figure 4.6	Diagrammatic Representation of $E_\kappa$ and $D_\kappa$ in $R_n^{2N}$ . . . . .	80
Figure 4.7	$(x, y, \kappa)$ Neighborhood Bubbles . . . . .	84

## CHAPTER 1. OVERVIEW

Software protection is one of the most important problems in the area of computing as it affects a multitude of players like software vendors, digital content providers, users, and government agencies. There are multiple dimensions to this broad problem of software protection. The most important ones are

- protecting software from reverse engineering.
- protecting software from tamper (or modification).
- preventing software piracy.
- verification of integrity of the software.

In this thesis we focus on these areas of software protection. The basic requirement to achieve these goals is to provide a *secure execution environment*, which ensures that the programs behave in the same way as it was designed, and the execution platforms respect certain types of wishes specified by the program. Industry as well as academic research is moving towards such an approach. Microsoft's heavy investment in a next generation trusted hardware platform (NGSCB) [42], the recent award by the U.S. Air Force Research Laboratory of a US\$1.8m research contract [18] involving software obfuscation, and the Trusted Computing Group [52] are samples of the research thrust in this area.

Several approaches have been researched in software-only solutions. Most notables ones are: **obfuscation through code transformations** [14]; **white-box cryptography** [13]; **software tamper resistance** [4]; and **software diversity** [15]. Any software-only solution to achieve *secure executing environment* seems to be inadequate. In the end, in most scenarios, it reduces to the problem of *last mile* wherein only if some small kernel of values could be isolated from the OS (as an axiom), the entire schema can be shown to work. Hence we take the approach of providing *secure execution environment* through architecture support. We exploit the power of reconfigurable components in achieving this.

The first problem we consider is to provide architecture support for *obfuscation*. This also achieves the goals of *tamper resistance*, *copy protection*, and *IP protection* indirectly. Earlier solutions to this problem used full-strength cryptographic primitives, and did not exploit the software specific properties. Our approach is based on the intuition that the software is a sequence of instructions (and data) and if the sequence as well the contents are obfuscated then all the required goals can be achieved. We identify *three dimensions* of information in a software and provide architectural support to obfuscate all these three dimensions. Chapter 2 presents a detailed description of the problem and our solution *Arc3D*.

The second problem we solve is integrity verification of the software particularly in embedded devices. Earlier solutions [2, 45] failed to identify the relationship between integrity verification problem and IP protection problem. Our solution is based on the intuition that an obfuscated (permuted) binary image without any dynamic traces reveals very little information about the IP of the program. Moreover, if this obfuscation function becomes a shared secret between the verifier and the embedded device then verification can be performed in a trustworthy manner. Chapter 3 explains the integrity verification problem and our solution *TIVA* in detail.

Cryptographic components form the underlying building blocks/primitives of any *secure execution environment*. Our use of reconfigurable components to provide software protection in both *Arc3D* and *TIVA* led us to an interesting observation about the power of reconfigurable components. Reconfigurable components provide the ability to use the *secret* (or key) in a much stronger way than the conventional cryptographic designs [40, 41]. This opened up an opportunity for us to explore the use of reconfigurable gates to build cryptographic functions.

Chapter 4 explains the proposed encryption, *REBEL* (REconfigurable Block Encryption Logic), design and its properties in detail. The advantages of *REBEL* are manifold. It is a completely new design paradigm, its hardware implementation is much faster when compared to the existing cryptographic functions, and it allows the use of a much larger secret (key) size than the block size. Since the future computing platforms essentially will contain some reconfigurable elements this design paradigm could be potentially beneficial. Even more importantly, *REBEL* seems to be more robust against cryptanalysis than the traditional block ciphers, since it uses the secrets more effectively.

## CHAPTER 2. ARC3D: ARCHITECTURE SUPPORT FOR 3D OBFUSCATION

Software obfuscation is defined as a transformation of a program  $\mathcal{P}$  into  $\mathcal{T}(\mathcal{P})$  such that the white-box and blackbox behaviors of  $\mathcal{T}(\mathcal{P})$  are computationally indistinguishable. However, robust obfuscation is impossible to achieve with the existing software only solutions. This results from the power of the adversary model in DRM which is significantly more than in the traditional security scenarios. The adversary has complete control of the computing node - supervisory privileges along with the full physical as well as architectural object observational capabilities. In essence, this makes the operating system (or any other layer around the architecture) untrustworthy. Thus the trust has to be provided by the underlying architecture. In this thesis, we develop an architecture to support 3-D obfuscation through the use of well known cryptographic methods. The three dimensional obfuscation hides the address sequencing, the contents associated with an address, and the temporal reuse of address sequences such as in loops (or the second order address sequencing). The software is kept as an obfuscated file system image statically. Moreover, its execution traces are also dynamically obfuscated along all the three dimensions of address sequencing, contents and second order address sequencing. Such an obfuscation makes it infinitesimally likely that good tampering points can be detected. This in turn provides with a very good degree of tamper resistance. With the use of already known software distribution model of ABYSS and XOM, we can also ensure copy protection. This results in a complete DRM architecture to provide both copy protection and IP protection.

### 2.1 Introduction

Digital rights management (DRM) deals with intellectual property (IP) protection and unauthorized copy protection. Software piracy alone accounted for \$13 billion annual loss [11] to the software industry in 2002. Software digital rights management traditionally consists of watermarking, obfuscation, and

tamper-resistance. All of these tasks are made difficult due to the power of adversary. The traditional security techniques assume the threat to be external. The system itself is not an adversary. This provides a *safe haven* or *sanctuary* for many security solutions. However, in DRM domain, the OS itself is not trustworthy. On the contrary, OS constitutes the primary and formidable adversary.

Any software-only solution to achieve DRM seems to be inadequate. In the end, in most scenarios, it reduces to the problem of *last mile* wherein only if some small kernel of values could be isolated from the OS (as an axiom), the entire schema can be shown to work. At this point, it is worth noting that even in the Microsoft's next generation secure computing base (NGSCB) [42], the process isolation from OS under a less severe adversary model is performed with hardware help. The NGSCB's goal is to protect the process from the OS corrupted by external attacks by maintaining a parallel OS look-alike called *nexus*. The *nexus* in turn relies upon a hardware Security Support Component (SSC) for performing cryptographic operations and for securely storing cryptographic keys.

The trusted computing group consisting of AMD, HP, IBM, and Intel among many others is expected to release trusted platform module (TPM) [52], to provide the SSC. The TPM is designed to provide such a root of trust for storage, for measurement, and for reporting. Hence, we believe that TPM provides building blocks for the proposed architecture. However, we identify additional capabilities needed to support robust 3D obfuscation. The proposed architecture obfuscation blocks can absorb TPM functionality (based on the released TPM 1.2 specifications [53]).

This chapter is organized as follows. Section 2.2 describes the obfuscation problem and its interaction with the existing cryptographic solutions. Section 2.3 discusses earlier proposed research and their drawbacks. Section 2.4 explains the basic building blocks of *Arc3D* and provides a high level overview. Section 2.5 provides operational details of *Arc3D* system. We describe various attack scenarios in Section 2.6. Section 2.7 gives the performance analysis of *Arc3D*. Section 2.8 concludes the chapter.

## 2.2 The Problem

In this section, we describe the problem we are addressing in detail. A software image is generated at the vendors site. This software image is then distributed to the customer through a transaction. Note that the software has to be generated in a standardized, well known, structure/format so that the OS can read it

and load it. The control flow sequence of the generated software instructions ought to be understandable by the CPU, even if the program image is obfuscated. The customer, who could be an adversary, has access to everything outside the CPU chip boundaries, including memory of the system and the OS. Additionally, from copy protection perspective, the vendor would like to associate the software only to the machine/CPU covered by the purchase transaction. The software should run only in its original form, *i.e.*, as provided by the vendor. Its tampered forms should not be executable.

The attributes that need to be supported by a DRM system are as follows.

1. Associability of Software to a particular CPU. (*copy-protection*)
2. Verifiability of the CPU's authenticity/identity. (*copy-protection, IP-protection*)
3. Binary file, conforming to a standardized structure, should not reveal any IP of the software through static analysis based reverse engineering. (*IP-protection – static obfuscation*)
4. Any modification of the binary file should make the software unusable. (*IP-protection – tamper-resistance*)
5. The program execution parameters visible outside CPU should not reveal any IP of the software. (*IP-protection – dynamic obfuscation*)

The first two problems are analogous to the real life problem of establishing trust between two parties followed by secret sharing on a secure encrypted channel. This is a well analyzed problem and solutions like Pretty Good Privacy (PGP) exist which uses a trusted Certification Authority (CA). This approach has been used in almost all the earlier research dealing with copy-protection ([56], [34]) and we too will use a similar approach. Note that the TPM specifications require the establishment of an endorsement key pair which is also registered with a CA. Furthermore, any number of attestation key pairs can be created to support trust between the CPU (with this TPM) and other clients (such as software vendors) which can also be registered with a certifying authority. The TPM specified protocol for creating an attestation identity also creates a shared secret between the TPM (or the corresponding CPU) and the client (software vendor). We expect to be able to use this capability of TPM or a similar protocol for copy protection.

The third problem requires prevention (minimization) of information leak from the static binary file/image. This could be viewed as the problem of protecting a message in an untrustworthy channel. One possible solution is to encrypt the binary file (the solution adopted by XOM [34] and ABYSS [56]). An alternative approach would recognize that the binary file is a sequence of instructions and data, with an underlying structure. Static obfuscation ([3, 16]) attempts to exploit a smaller subset of these program level structure attributes.

The fourth problem requires the binary image to be tamper resistant. Any modifications to the binary image should be detectable by the hardware. Message Digest, which is a one-way hash of the message could be used to solve this problem. This once again is a generic solution which is applicable to any message transaction that does not use any specific properties of a binary image. We rely upon obfuscation to provide the tamper-resistance in the following way. Tampering gains an advantage for the adversary only if the properties of the tampering point - the specific instruction or data at that point - are known. However, obfuscation prevents the adversary from associating program points with specific desirable properties (such as all the points that have a branch, call sites, to a specific procedure or all the data values that point to a specific address). Hence most tampering points are randomly derived resulting in the disabling of the program, which we do not consider to be an advantage to the adversary in the DRM model where the adversary/end user has already purchased rights to disable the program.

The fifth problem dictates that the CPU not trust anything outside its own trusted perimeter including any software layer. The problem is simplified by the fact that CPU can halt its operations once it detects any untrustworthy behavior. The attributes of the application program execution trace space, which the CPU has to protect, can be thought of as having three dimensions, namely, instructions (content), addresses at which the instructions are stored (address sequencing), and the temporal sequence of accesses of these addresses (second-order address sequencing). All of these three dimensions have to be protected in order to prevent any information leakage. This holds true even for data.

## 2.3 Previous Research

### 2.3.1 ABYSS

ABYSS [56] is an architecture for protecting the application software. It can be used as a uniform security service across a range of computing systems. The system contains both *protected* and *unprotected* processes. Protected processes are executed in a *protected processor*. Protected processor constitutes a minimal, but complete, computing system. It has sufficient memory to store protected parts of the application. It also has non-volatile storage space to store *Rights-To-Execute* (RTE) attributes of an application. There exists a *supervisor process* which ensures the logical and procedural security of the protected processor.

Supervisor process handles the decryption of protected process from disk and its loading into RAM. It also isolates the applications from each other, and from their unprotected parts. “Rights to execute” contains the privileges of the application and a “Key” to decrypt the application. *One-Use* tokens are used to authorize the installation of RTE. The encryption model used is shown in Box 2.1. Where,  $K_A$  is the Application key, decided by the software vendor and can be unique for each copy or common for multiple copies.  $K_S$  is the Supervisor key and is the shared secret between software vendor and hardware manufacturer. Drawbacks of ABYSS include non-scalability and unexplained OS interactions.

$$\begin{array}{c}
 K_A \{ \textit{Protected part of application} + \textit{Tok} \} \\
 K_S \{ \textit{RTE} + K_A \}
 \end{array}
 \tag{2.1}$$

### 2.3.2 TrustNo1 Cryptoprocessor

TrustNo1 [32] proposes hardware, firmware, OS and key management mechanisms necessary to apply the cryptoprocessor concept in multitasking OS systems. It contains a protected on-chip memory key table that can store segment keys. This key table can only be accessed by firmware that is also stored in protected on-chip memory. Segment descriptors are extended by a new field, which contains the index of the key used to decrypt it. Each cache line is encrypted with the segment’s key and some of the MSB bits of cache line address. This makes each key unique and hence prevents the cipher instruction search attack. The memory manager grants any access only if it originates from an instruction in the same segment. This

prevents even OS from reading or modifying the cleartext of an encrypted segment and from calling parts of the code in an uncontrolled fashion. Box 2.2 lists the special operations supported by the cryptoprocessor.

*save\_state* → store CPU state

*restore\_state* → restore CPU state

*transfer\_state* → copy current CPU state

*supervisor\_call* → execute system calls

(2.2)

### 2.3.3 XOM

XOM [34] is based on compartmentalized - a process in one compartment cannot access data from another compartment - machine. Application is encrypted with a symmetric key which in turn is encrypted with public asymmetric key of XOM. All levels of memory (L2, L1, Registers) are tagged with the session key ID, specific to the process that is running. XOM stores session keys in the key table. The basic operations exported by XOM are *enter\_xom*, *exit\_xom*, *mv\_to\_null*, *mv\_from\_null*, *save\_secure*, *restore\_secure*, *load\_secure*, *store\_secure*, *load\_from\_null*, and *store\_to\_null*. Using these an OS can manage any protected process and all the normal operations in an XOM node, except *fork* of a protected process.

### 2.3.4 HIDE

HIDE [59] is an extension of XOM. It points out the fact that XOM does not protect the third dimension of the information space, *i.e.*, the time order of the address trace. Hence even if the instructions (and data) themselves are obfuscated (through encryption in XOM), the address trace gives the adversary power to deduce the control flow graph (CFG). HIDE further argues that the information obtained from the CFG could lead to serious security breaches of the software. HIDE provides a solution to this address bus leakage problem consisting of chunk-level protection with hardware support and a flexible interface. Compiler optimizations and user specifications could be further utilized to deploy the underlying hardware solution more efficiently to provide better security guarantees.

### 2.3.5 Oblivious RAM

Goldreich and Ostrovsky [26] offered one of the first schemes for software protection. They extended the oblivious Turing machine model to oblivious RAMs. An oblivious RAM presents a memory access footprint that does not depend on the program input. This prevents information leak about the program CFG. Oblivious RAM requires  $O(\sqrt{m})$  steps for each original memory access. This is a very high overhead for present day architectures.

### 2.3.6 Dallas Semiconductor

DS5002FP [20] is a secure 8-bit microcontroller from Dallas Semiconductor which uses bus-encryption. The DS5002FP implements three on-chip block-cipher functions  $E_A$  for 17-bit address-bus encryption,  $E_D$  for 8-bit data-bus encryption, and  $E_D^{-1}$  for 8-bit data-bus decryption. The encryption functions are fixed unless changed by uploading a new secret key. This allows the adversary to build up information by running the program multiple times and observing its behavior. Kuhn [33] proposed such an attack to extract the secrets stored in DS5002FP microcontroller.

### 2.3.7 Obfuscation

We use the term *obfuscation* in a different way than the earlier literature ([14, 36]). We refer to any obfuscation mechanism that hides the control flow from static analysis through a binary image or CFG transformation as *static obfuscation*. However, these obfuscation mechanisms cannot defeat an adversary with access to architecturally visible parameters such as memory address bus. Our use of term *obfuscation* refers to schemes that hide dynamic execution model parameters as well. Note that in our obfuscation model, even the instructions themselves can be transformed into potentially invalid ones.

### 2.3.8 Analysis

Almost all of the earlier research, except HIDE, does not hide the temporal sequencing of memory accesses. Neither do these solutions exploit the software specific properties. The solution proposed by HIDE to prevent information leak through the address and memory bus is weak. This is because the adversary can see the contents of the memory before and after an address-permutation. It is possible

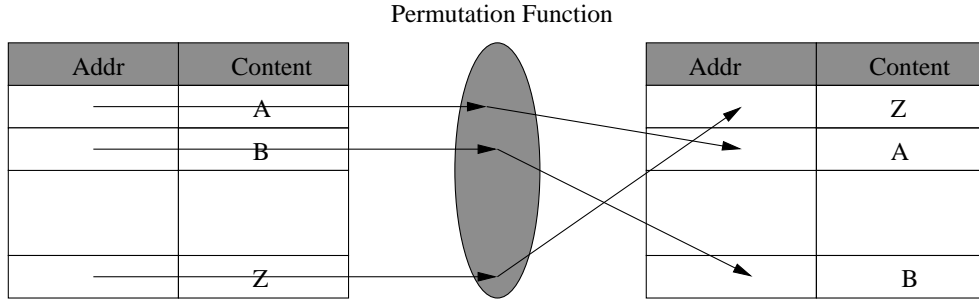


Figure 2.1 Weakness of HIDE approach

because the encryption function applied to the contents is not address dependent. Hence, for instance, if the contents at two distinct addresses  $A_i$  and  $A_j$  are also distinct  $C_{A_i}$  and  $C_{A_j}$  then the following information leak path exists. For a program sequence within a loop, when instructions reoccur at the address and instruction buses, HIDE permutes the addresses within a page for the second (or subsequent) iteration. If  $A_i$  is permuted to a new address  $\pi(A_i)$  the contents at  $\pi(A_i)$  would still appear as  $C_{A_i}$ . Hence a simple comparison would be able to determine the permutation  $\pi$ . Figure 2.1 illustrates this fact. Thus it takes only  $\frac{N(N+1)}{2}$  comparisons to reverse-engineer the permutation, where  $N$  is the permutation size. Assuming that there are 1024 cache-blocks in a page, the strength of such a permutation is less than  $2^{20}$ . Even in the chunk mode, which performs these permutations in a group of pages, the complexity grows only linearly, and hence could be easily broken.

The proposed architecture *Arc3D* addresses all these issues. Moreover, computational efficiency of proposed methods is a key criterion for inclusion in *Arc3D*. We make use of software structure to provide obfuscation and tamper-resistance efficiently.

## 2.4 Proposed Architecture: Arc3D

The overall *Arc3D* architecture is shown in Figure 2.2. The main affected components of the microarchitecture are the ones that handle virtual-addresses. These components include the translation lookaside buffer (TLB) and page table entries (PTE). We first describe the objectives of the obfuscation schema.

### 2.4.1 Obfuscation Schema

The goal of obfuscation is to remove the correlation between

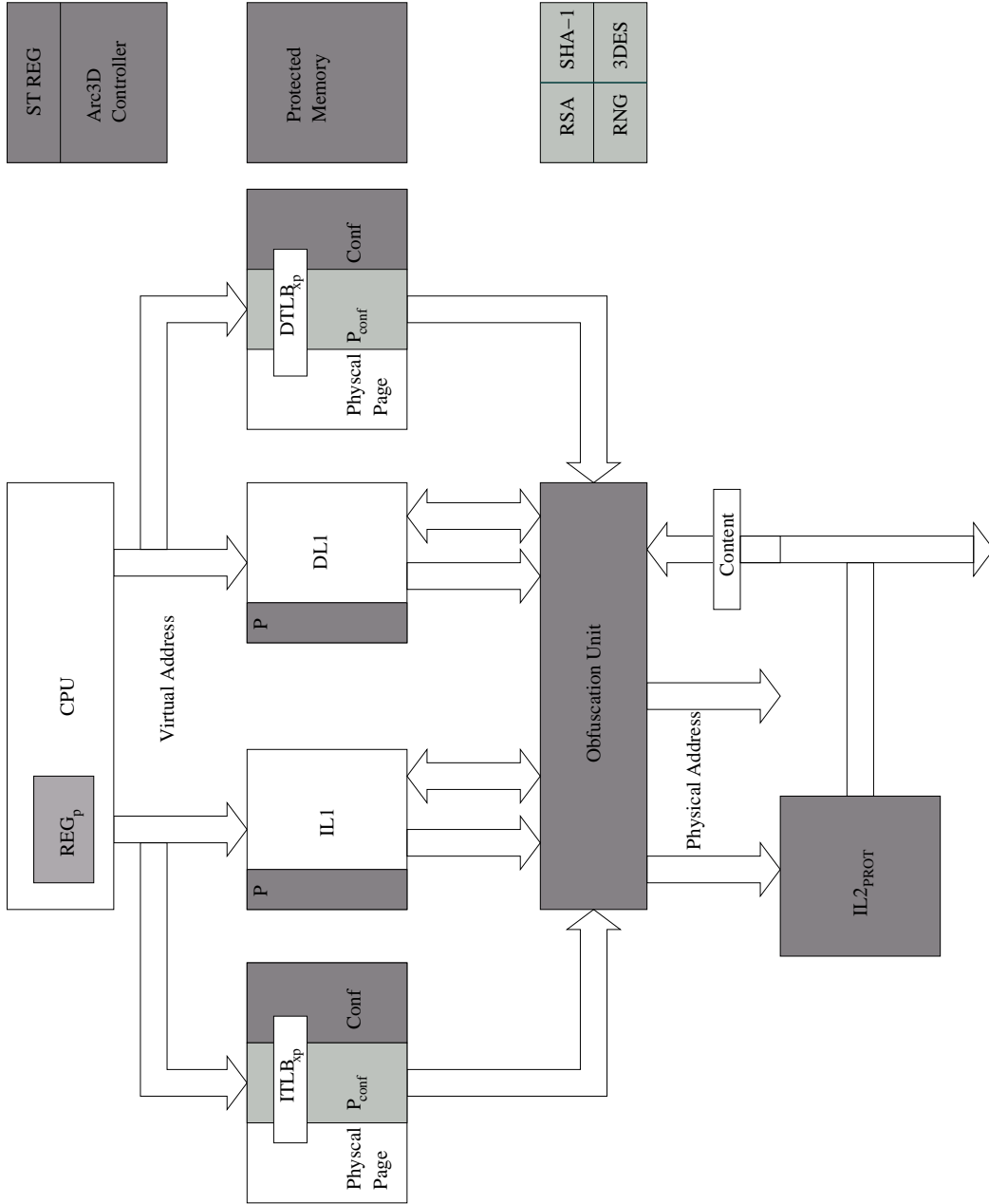


Figure 2.2 Overall Schema of Arc3D Architecture

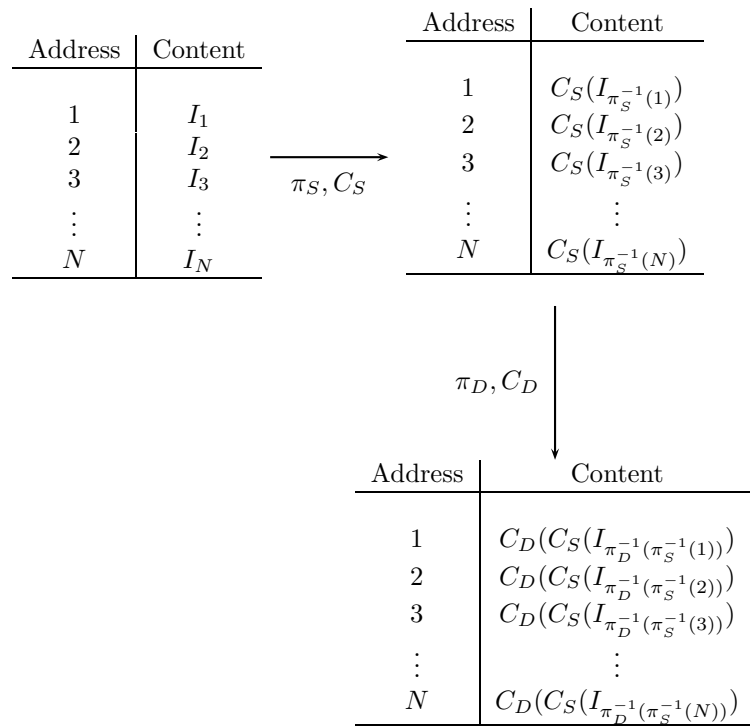


Figure 2.3 Static and Dynamic obfuscation

1. the CFG and the static binary image.
2. the static binary image and the dynamic execution image.

Traditional static obfuscation techniques try to obscure disassembling and decompilation stages to remove the correlation between the static image and the CFG. But these techniques are transparent to architecture and do not remove the correlation between the static image and the dynamic execution image. Thus an adversary monitoring the address traces could very well extract the CFG.

We use *architecture aware* obfuscation of both *sequence* and *content* to achieve this goal. A program is obfuscated both statically and dynamically. The objective of obfuscation is to permute the address sequence and to hide the contents (so that the mapping from an address  $A_i$  to its contents  $C_{A_i}$  is not obvious). Let  $V$  denote the virtual address sequence  $\{0, 1, 2, \dots, 2^{32} - 1\}$  for a 32-bit architecture. The classical static binary image layout maps instructions in the order:  $I_0, I_1, \dots, I_{2^{32}-1}$ . In other words, contents  $I_j$  are mapped in the virtual address sequence  $j \in V$ . A static address permutation function  $\pi_S$  can be applied to the binary image so that the instructions in the static binary image appear in the following order:  $I_{\pi_S^{-1}(0)}, I_{\pi_S^{-1}(1)}, I_{\pi_S^{-1}(2)}, \dots, I_{\pi_S^{-1}(2^{32}-1)}$ . The static address permutation function disperses the  $j$ th instruction in the virtual address sequence  $V, I_j$ , into the static sequence number  $\pi_S(j)$ .

The contents also need to be obfuscated. A content obfuscation function  $C_S(I_j, j)$  transforms the original contents  $I_j$ . Note the dependence of the content obfuscation function on the address/sequence number of the instruction  $j$  as well. This makes sure that the contents  $I$  (an instruction encoded as  $I$ ) will look different when mapped to two distinct sequence numbers  $j$  and  $k$ . Hence the program will appear to be in the following sequence after static obfuscation has been applied by the software vendor:  $C_S(I_{\pi_S^{-1}(0)}, 0), C_S(I_{\pi_S^{-1}(1)}, 1), C_S(I_{\pi_S^{-1}(2)}, 2), \dots, C_S(I_{\pi_S^{-1}(2^{32}-1)}, 2^{32} - 1)$ .

The dynamic execution of the program will need to know both the content obfuscation function  $C_S$  and the address permutation function  $\pi_S$ . When an instruction in the virtual address sequence  $j \in V$  needs to be fetched, the processor would have to issue an address  $\pi_S(j)$ . Let memory return  $M[\pi_S(j)]$  in response to this read. Now the processor would have to deobfuscate the contents as  $C_S^{-1}(M[\pi_S(j)], j)$ . This schema sums up (and generalizes) most permutation based obfuscation schemes. The problem with this approach though is that an adversary watching the address bus can infer the correct address sequence for the instructions since both  $j$  and  $\pi_S(j)$  are known for many (or all) instantiated addresses  $j \in V$ .

Goldreich et al. [26] provide a theoretical solution to this problem which is applicable to a software-hardware (SH) package. Specifically, it seems to assume a lightweight, embedded operating system which is not necessarily controlled by the owner of the SH-package.

We propose to perform yet another level of dynamic obfuscation so that address bus visible address sequence is yet another permutation of the virtual address sequence. Another pair of address permutation and content obfuscation functions which are dynamically chosen,  $\pi_D$  and  $C_D(I_j, j)$ , help achieve dynamic obfuscation. Hence the static image sequence  $C_S(I_{\pi_S^{-1}(0)}, 0)$ ,  $C_S(I_{\pi_S^{-1}(1)}, 1)$ ,  $C_S(I_{\pi_S^{-1}(2)}, 2)$ ,  $\dots$ ,  $C_S(I_{\pi_S^{-1}(2^{32}-1)}, 2^{32}-1)$  is actually loaded in the memory as  $C_D(I_{\pi_D^{-1}(0)}, 0)$ ,  $C_D(I_{\pi_D^{-1}(1)}, 1)$ ,  $C_D(I_{\pi_D^{-1}(2)}, 2)$ ,  $\dots$ ,  $C_D(I_{\pi_D^{-1}(2^{32}-1)}, 2^{32}-1)$ . Hence it is important that the loading and storing (specifically virtual address translation) function be taken away from the operating system and be part of a trusted component within the architecture. This trusted component is also responsible for guarding the program secrets  $C_S$ ,  $C_D$ ,  $\pi_S$ ,  $\pi_D$ . Figure 2.3 shows the obfuscation schema in detail.

## 2.4.2 Overall Schema

As stated earlier, Figure 2.2 shows the global schema for the proposed architecture. The shaded areas are the additional components of *Arc3D* over the base architecture. Shading hues also indicate the access rights as follows. The lightly shaded areas contain information accessible to the outside world, *i.e.*, OS. The darkly shaded areas contain secret information accessible only to *Arc3D*. *Arc3D* has two execution modes, namely *protected* and *unprotected* mode. It has a protected register space  $REG_p$  which is accessible only to a protected process.

The core of *Arc3D* functionality is obfuscation. It is achieved by modifying the virtual-address translation path - translation look aside buffer (TLB) - of the base architecture. In addition to holding the virtual-address to physical-address mapping, page table entry (PTE), the TLB has the obfuscation configuration ( $P_{conf}$ ). This  $P_{conf}$  is essentially the shared secrets  $C_S$ ,  $C_D$ ,  $\pi_S$ ,  $\pi_D$  in encrypted form. In order to avoid frequent decryption, *Arc3D* stores them in decrypted form in  $Conf$  section of  $TLB_{xp}$ . This section of TLB is updated whenever a new PTE is loaded into  $TLB_{xp}$ . *Arc3D* assumes parallel address translation paths for data and instructions, and hence Figure 2.2 shows DTLB and ITLB separately.

The address translation for a protected process occurs in the obfuscation unit. Sections 2.4.4 and 2.4.5

explain in detail the address sequence and content obfuscation algorithms respectively. *Arc3D* uses the same logic for both static and dynamic obfuscations. The basis of these obfuscations is the permutation function which is explained in Section 2.4.3. *Arc3D* has a protected L2 cache, which is accessible only to a protected process, thus providing temporal order obfuscation.

*Arc3D* controller provides the following interfaces (APIs) which enable the interactions of a protected process with the OS.

1. *start\_prot\_process*: Allocate the necessary resources and initialize a protected process.
2. *exit\_prot\_process*: Free the protected resources allocated for the current protected process.
3. *ret\_prot\_process*: Return to the current protected process from an interrupt handler.
4. *restore\_prot\_process*: Restore a protected process after a context switch.
5. *transfer\_prot\_process*: Fork the current protected process.

These APIs and their usage are explained in detail in Section 2.5.

### 2.4.3 Reconfigurable Bijective Function Unit

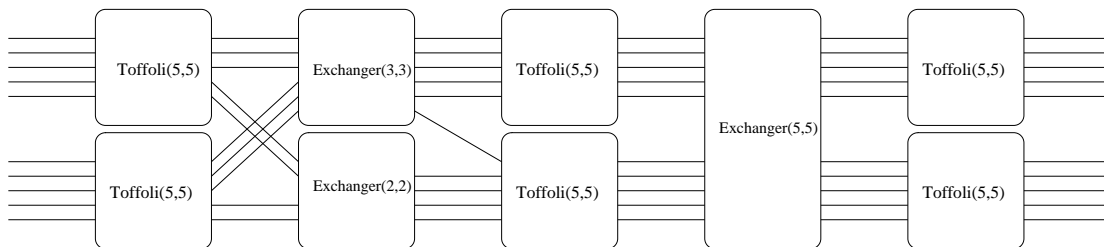


Figure 2.4 Reconfigurable Bijective Obfuscation Unit

Obfuscation unit is a major component of *Arc3D*. This unit is responsible for generating *bijection* functions  $\pi$ . There are  $2^n!$  possible  $n$ -bit reversible functions. Reconfigurable logic is well-suited to generate a large dynamically variable subset of these reversible functions. Figure 2.4 shows one such schema for permutation of 10 address bits (specifying a page consisting of 1024 cache-blocks). Before explaining the blocks of Figure 2.4, we observe that there are  $\left(2^{2^n}\right)^n$  possible functions implemented in a

$n \times n$  look up table (LUT) or  $n$   $n$ -LUTs. But only a subset of them are bijective. We wish to implement only reversible (conservative) gates ([24], [5]) with LUTs.

**Definition 2.4.1.** A Toffoli gate,  $Toffoli(n,n)(C,T)$ , is defined over a support set  $\{x_1, x_2, \dots, x_n\}$  as follows. Let the control set  $C = \{x_{i_1}, x_{i_2}, \dots, x_{i_k}\}$  and the target set  $T = \{x_j\}$  be such that  $C \cap T = \emptyset$ . The mapping is given by

$$Toffoli(n,n)(C,T)[x_1, x_2, \dots, x_n] = [x_1, x_2, \dots, x_{j-1}, z, x_{j+1}, \dots, x_n]$$

where  $z = x_j \oplus (x_{i_1} \wedge x_{i_2} \wedge \dots \wedge x_{i_k})$ .

**Definition 2.4.2.** Fredkin gate,  $Fredkin(n,n)(C,T)$ , is defined over a support set  $\{x_1, x_2, \dots, x_n\}$  as follows. Let the control set  $C = \{x_{i_1}, x_{i_2}, \dots, x_{i_k}\}$  and the target set  $T = \{x_j, x_l\}$  be such that  $C \cap T = \emptyset$ . The mapping is given by

$$Fredkin(n,n)(C,T)[x_1, x_2, \dots, x_n] = [x_1, x_2, \dots, x_{j-1}, p, x_{j+1}, \dots, q, \dots, x_n]$$

where  $k = x_{i_1} \cdot x_{i_2} \cdot \dots \cdot x_{i_k}$ ,  $p = (x_j \cdot \bar{k}) + (x_l \cdot k)$ , and  $q = (x_j \cdot k) + (x_l \cdot \bar{k})$ .

Both Fredkin [24] and Toffoli [51] have defined classes of reversible gates. We use  $Toffoli(5,5)$  gates with 5-input bits and 5-output bits in our scheme as shown in Figure 2.4. However, we could easily replace them by  $Fredkin(5,5)$  gates. The domain of configurations which can be mapped to each of the LUTs consists of selections of sets  $T$  and  $C$  such that  $T \cap C = \emptyset$ . For a support set of 5 variables, the number of unique reversible Toffoli functions is  $4 \binom{5}{1} + 3 \binom{5}{2} + 2 \binom{5}{3} + \binom{5}{4}$ . Each of these terms captures control sets of size 1, 2, 3, and 4 respectively. Ignoring control sets of size 1, we get a total of 55 reversible functions. Thus total permutation space covered by all six of these gates is  $(55)^6 \approx 2^{34}$ . There are several redundant configurations in this space. We estimate this redundancy later in this section.

The exchanger blocks shown in Figure 2.4 perform a *swap* operation. It has two sets of inputs and two sets of outputs. The mapping function is  $S_{ok} = S_{ik}$  if  $X = 0$ , and  $S_{ok} = S_{i\bar{k}}$  if  $X = 1$ , where,  $S_{ik}$  is the input set,  $S_{ok}$  is the output set,  $X$  is configuration bit, and  $k$  is 0 or 1. Since *exchange* is also bijective, the composition of *Toffoli* gates and *exchangers* leads to a bijective function with large population diversity. Other interesting routing structures may also guarantee bijections. But a typical FPGA routing matrix configuration will require extensive analysis to determine if a given routing configuration is bijective. One

point to note here is that we chose to implement a 10 *bit* permutation function with *Toffoli(5,5)* gates instead of a direct implementation of *Toffoli(10,10)*. This is because an  $n$ -LUT requires  $2^n$  configuration bits and hence 10-LUTs are impractical in the reconfigurable computing world.

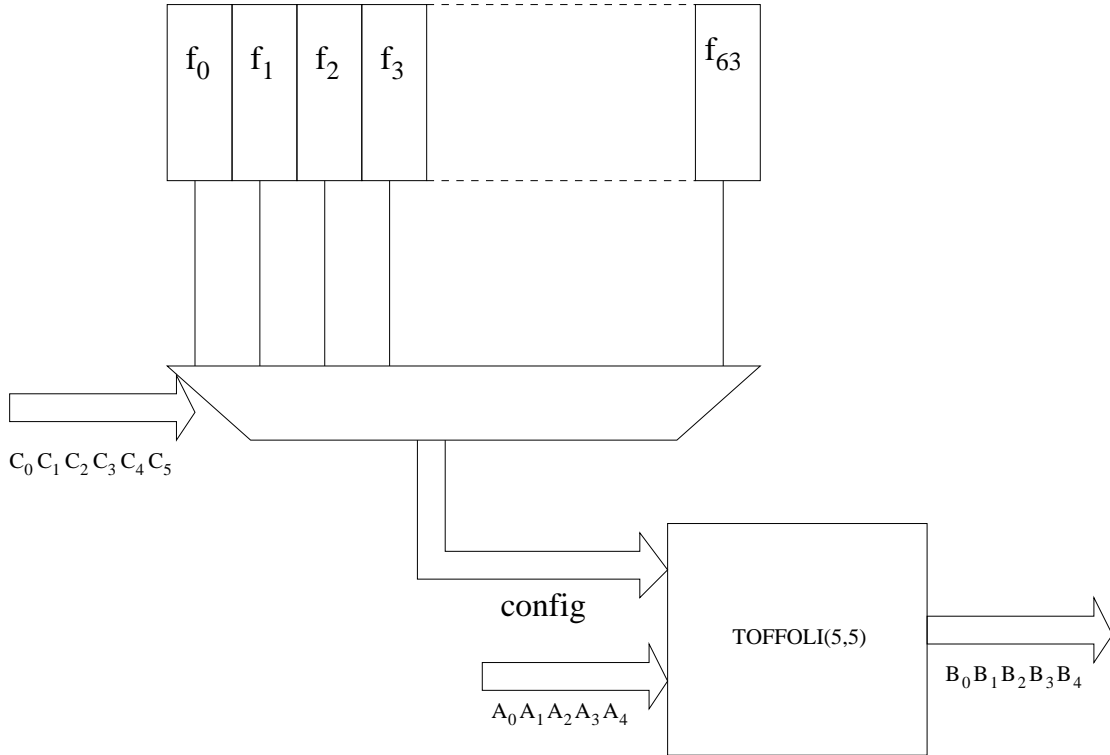


Figure 2.5 Configuration Selection for each LUT

Having fixed the reconfigurable logic to perform the obfuscation (permutation), we need to develop a schema for the LUT configuration. A simple mechanism would be to store all the 55 possible configurations at each of the LUTs (similar to DPGA of DeHon [21]). In addition to 4 *input bits*, each LUT will also have 6 *configuration bits* to choose from one of the 55 configurations (assuming some configurations are repeated to fill the 64 locations), as shown in Figure 2.5. In Figure 2.5  $A_0 A_1 A_2 A_3 A_4$  represent the input address bits,  $f_0, f_1, f_2, \dots, f_{63}$  represent the 64 configurations, and  $C_0 C_1 C_2 C_3 C_4 C_5$  represent the configuration bits. Each of the *exchanger* blocks also requires 1 configuration bit. Thus a total of 39 configuration bits are needed by the reversible logic of Figure 2.4.

### 2.4.3.1 Estimating Redundancy in Configurations

The most reasonable and efficient way to generate configurations is to generate each configuration bit independently and randomly. However this process may generate two configurations that represent the same mapping (from incoming block# to the outgoing block#). Such aliasing reduces the diversity of the address mapping functions making them more predictable to the adversary. We capture the degree of this aliasing with the concept of *redundancy level* of a reconfigurable obfuscation circuit. The redundancy level can be defined as the fraction of  $2^{39}$  configurations that alias (generate a repeated, non-unique mapping function).

We assessed the redundancy level of the address permutation schema in Figure 2.4 through the following setup. We simulated this FPGA circuit with  $2^{20}$  randomly generated configurations. For each of these configurations, we derived the corresponding bijective function by exercising all the 10-bit inputs sequences. Each unique bijective function was stored. When a bijective function  $f_i$  from a new random sequence from the  $2^{20}$  runs is encountered, it is compared against all the stored bijective functions that have already been generated. At the end of  $2^{20}$  runs, we end up with  $k \leq 2^{20}$  unique functions  $f_i$  for  $0 \leq i \leq k$  and their redundancy count  $r_i$  (function  $f_i$  occurs in  $r_i$  of the  $2^{20}$  runs). The *redundancy level* is computed as  $[\sum_{r_i > 1} 1] / 2^{20}$ . We repeated this experiment several times in order to get a statistical validation of our experiment. All the values are listed in Table 2.1.

This experiment can be modeled as a random experiment where we have  $N(=2^{39})$  balls in a basket which are either *red(=redundant)* or *green(=non-redundant)*. We need to estimate the number of red balls in the basket by picking  $n(=2^{20})$  balls where all the balls are equally likely. We define a random variable  $X$  such that  $X = 1$  if the chosen ball is *red* and  $X = 0$  otherwise. The mean of such a random variable is nothing but the redundancy level. We see from Table 2.1 that the mean is close to zero ( $\approx 0.3\%$ ) and hence the variance is equal to the mean. Using the variance and mean we estimated the 99% confidence interval of the mean of  $X$ , *i.e.*, the average redundancy level of reconfigurable obfuscation circuit. From the table, it is clear that with probability 0.99 the average percentage of redundant configurations will lie within 0.3058 to 0.3444, *i.e.*, only 3 out of 1000 randomly generated configurations will be redundant.

Table 2.1 Redundancy Estimation (# of Random Configs =  $2^{20}$ )

Random Seed	# of Redundant Functions	Avg % Redundancy	99% CI
89ABCDEF	3359	0.320	0.3058 to 0.3342
11223344	3409	0.325	0.3107 to 0.3393
12345678	3441	0.328	0.3136 to 0.3424
34567890	3417	0.325	0.3107 to 0.3393
789012345	3469	0.330	0.3156 to 0.3444
8901234567	3460	0.330	0.3156 to 0.3444
56789012345	3460	0.330	0.3156 to 0.3444

#### 2.4.4 Obfuscating the Sequence

We can use the reconfigurable permutation unit defined in Section 2.4.3 to achieve sequence obfuscation. Note, however, that even though we have shown the circuit for a 10-bit permutation, the methodology is applicable to an arbitrary number of address bits. We believe that at least 10 address bits need to be permuted in order to have a reasonably large permutation space. The choice of 10-bits is also dictated by the structure of the software. Software objects, both instruction and data, are viewed by the architecture in various granularities. The RAM memory resident objects are viewed in the units of *pages*. The cache resident objects on the other hand are viewed in the units of *blocks*. This argues for the obfuscation boundaries defined by these units. Hence we obfuscate the sequence of *cache-blocks* within a page. Any sequence obfuscation within *page* level needs to interact with the page management module of the OS. If the obfuscated sequence crosses the *page* boundary, the permutation function ( $\pi$ ) has to be exposed to the OS. This is the reason why we cannot obfuscate sequences of pages. In the other direction, permuting the sequences of sub-units of cache-blocks seriously affects the locality of cache resulting in severe performance degradation. Moreover, since the contents of a cache-block are obfuscated, the information leak through the preserved, original sequence of cache sub-blocks is minimized. Considering a *page* size of 64KB with 64B *cache-blocks*, as is the case with Alpha-21264, we get 1024 *cache-blocks* per page, *i.e.*, 10-bits of

obfuscation.

#### 2.4.5 Obfuscating the Contents

In cryptography, the *one time pad* (OTP), sometimes known as the *Vernam cipher*, is a theoretically unbreakable method of encryption where the plaintext is transformed (for example, XOR) with a random *pad* of the same length as the plaintext. The structure of the software objects determines the protection granularities once again. We can consider a program as a sequence of fixed sized messages, *i.e.*, *cache-blocks*. If we have unique OTPs for each one of the cache-blocks in the software, the contents are completely protected. However, the storage and management of that many OTPs is highly inefficient. Nonetheless, we at least have to guarantee that every *cache-block* within a *page* has a unique OTP. This is to overcome the weakness in HIDE (as explained in Section 2.3, Figure 2.1). If the adversary-visible contents of the memory locations are changed after each permutation (as with unique cache-block OTP per page), then  $n$ -bit permutation is  $2^n!$  strong. This is in contrast with the strength of the order of  $2^n$  exhibited by the original HIDE scheme.

In order to provide a unique OTP per cache-block per page, one option is to generate a random OTP mask for each cache-block for each page. A more efficient solution, however, is to pre-generate  $N_b$  OTPs for every cache-block within a page ( $OTP[b_i]$  masks for  $0 \leq b_i < N_b$  for a cache with  $N_b$  blocks). However, the association of an OTP with a cache-block is randomized with the  $\pi_c$  function. The  $\pi_c$  function can be chosen differently for each page to provide us with a unique OTP per cache-block. This simplifies the hardware implementation of content obfuscation unit as well, since each page is processed uniformly in this unit except for the  $\pi_c$  function. Hence a program image will need to provide a page of OTPs which will be used for all its pages. It also needs to specify a unique mapping function  $\pi_c$  per page. Since we already have the reconfigurable permutation logic of Section 2.4.3 in *Arc3D*, we can use it to implement the function  $\pi_c$  as well. This results in 39-bits per page overhead for the specification of the content obfuscation. Note that the OTP based content encryption can be easily replaced by any other *bijective* function.

### 2.4.6 Obfuscating Temporal Order (Second-Order Address Sequences)

The second-order address sequences are derived from iterative control constructs within a program. Consider a loop of  $k$  instructions which is iterated  $N$  times. The expected address sequence in such an execution is  $\{I_{(0,0)}, I_{(1,0)}, \dots, I_{(k-1,0)}\}, \{I_{(0,1)}, I_{(1,1)}, \dots, I_{(k-1,1)}\}, \dots, \{I_{(0,N-1)}, I_{(1,N-1)}, \dots, I_{(k-1,N-1)}\}$  where  $I_{i,j}$  denotes the  $i^{\text{th}}$  instruction in the loop body in the  $j^{\text{th}}$  loop iteration. In this sequence, if an adversary is able to tag the boundaries of loop iteration, a strong correlation exists between successive iteration traces:  $\{I_{(0,l)}, I_{(1,l)}, \dots, I_{(k-1,l)}\}$  and  $\{I_{(0,l+1)}, I_{(1,l+1)}, \dots, I_{(k-1,l+1)}\}$ . In fact, instruction  $I_0$  occurs in the same relative order from the loop sequence start point in both (or all) the iterations. This allows an adversary to incrementally build up information on the sequencing. Whatever sequence ordering is learnt in iteration  $l$  is valid for all the other iterations. The second-order address sequence obfuscation strives to eliminate such correlations between the order traces from any two iterations.

Interestingly, the second-order address sequence obfuscation is an inherent property of a typical computer architecture implementation. The access pattern we observe outside the CPU is naturally obfuscated due to various factors like *caching*, *prefetching*, and several other *prediction* mechanisms aimed at improving the performance. But these architecture features are also controllable, directly or indirectly, by the OS and other layers of software. For example, the adversary could flush the cache after every instruction execution. This renders the obfuscation effect of *cache* non-existent. To overcome such OS directed attacks, it is sufficient to have a reasonably sized *protected-cache* in the architecture which is *privileged* (only accessible to secure processes). We expect a *cache* of the same size as a *page*, in our case 64KB, should be able to mask the effects of loops. Encrypted or content-obfuscated *cache-blocks* already obfuscate CFGs (within the cache-block). This is because a 64B cache-block contains 16 instructions if we assume instructions of length 32-bits.

## 2.5 Arc3D in Operation

We have developed and described all the building blocks of *Arc3D* in Section 2.4. In this section, we explain its operation with respect to the software interactions in detail, from software distribution to the management of a protected process by the OS using the APIs provided by *Arc3D*.

### 2.5.1 Software Distribution

*Arc3D* provides both *tamper-resistance* and *IP-protection* through obfuscation. Hence, a software vendor should be able to obfuscate the static image of a binary executable. Moreover, a mechanism to distribute the *static obfuscation* configuration from the vendor to *Arc3D* needs to be supported. This configuration constitutes the shared secret between the vendor and *Arc3D*. Trust has to be established between *Arc3D* and the vendor in order to share this secret. Once the trust is established, the binary image along with the relevant configuration can be transferred to *Arc3D*.

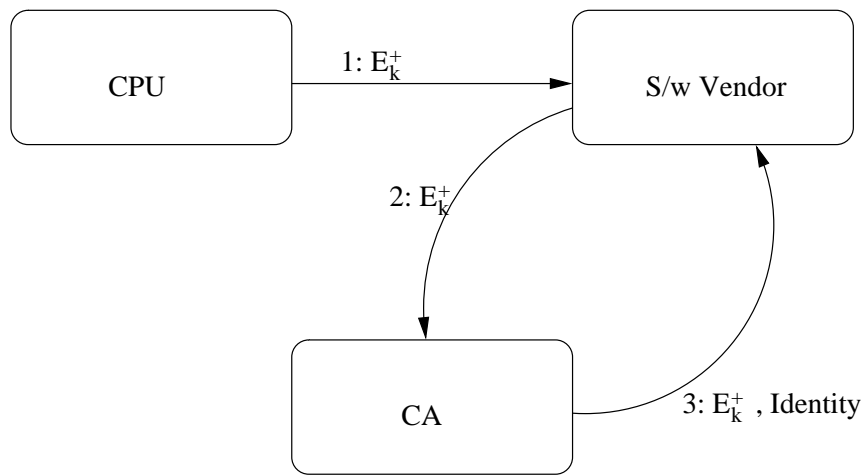


Figure 2.6 Three Party Trust Model

#### 2.5.1.1 Trust Establishment

We assume that there exist protected elements within the CPU which are accessible only to the architecture, and not to any other entities. We also assume that every CPU has a *unique identity*, namely, its *public-private key pair*  $(E_k^+, E_k^-)$ . This key pair is stored in the protected space of the CPU. A TPM's endorsement key pair constitutes such an identity. Public part of this key pair,  $E_k^+$ , is distributed to a *certification authority* (CA). CA verifies the CPU vendor's authenticity, associates  $E_k^+$  with CPU vendor's identity and other information (such as model number, part number, etc.). Any party entering a transaction with the CPU (such as a software vendor) can query the CA with  $E_k^+$  in order to establish trust in the CPU. Since CA is a trusted entity, the data provided by CA can also be trusted. This is very similar to the PGP model of trust establishment and is shown in Figure 2.6.

An important point to note here is that the trust establishment and the key management mechanisms do not constitute the crux of *Arc3D* architecture. *Arc3D* could use any model/policy for this purpose. We use this model for illustration purposes only. It could very well be adapted to use the TPM [52] model.

### 2.5.1.2 Binary Image Generation

Software vendor receives  $E_k^+$  from the CPU. It queries the CA to derive the architecture level specifications of the CPU, relevant for static obfuscation which include details such as *cache-block* size, minimum supported *page* size. Software vendor generates the binary file targeted at the appropriate cache-block and page sizes. It generates two sets of random configurations per page. One configuration is to obfuscate the sequence of *cache-block* addresses within a page ( $\pi_s$ ) and the second configuration is to obfuscate the association of OTPs with *cache-block* address ( $\pi_{c_s}$ ). The content obfuscation requires the software vendor to further generate a page sized OTP ( $OTP_s, OTP_s[b_i]$  for all  $0 \leq b_i < N_b$ ). These functions can then be used along with the FPGA obfuscation unit in a CPU or with a software simulation of its behavior to generate the obfuscated binary file.

---

Algorithm 1 Page Obfuscation Function: *page\_obfuscate*

**Required Functions**

$F_{obf}(conf\_sel, addr) \Leftarrow$  Reconfigurable Obfuscation Unit

**Inputs**

$OTP_{arr} \Leftarrow$  array of OTP

$page_i \Leftarrow$  input page

$conf_{seq} \Leftarrow$  conf\_sel for sequence obfuscation

$conf_{cont} \Leftarrow$  conf\_sel for content obfuscation

$N_b \Leftarrow$  number of *cache blocks* in a page

**Outputs**

$page_o \Leftarrow$  output of page

**Function**

**for**  $k = 0$  to  $N_b - 1$  **do**

$out = F_{obf}(conf_{seq}, k)$

$l = F_{obf}(conf_{cont}, k)$

$OTP = OTP_{arr}[l]$

$page_o[out] = page_i[k] \oplus OTP$

**end for**

---

Although this kind of obfuscation is applicable to any binary image, the software vendor could enforce

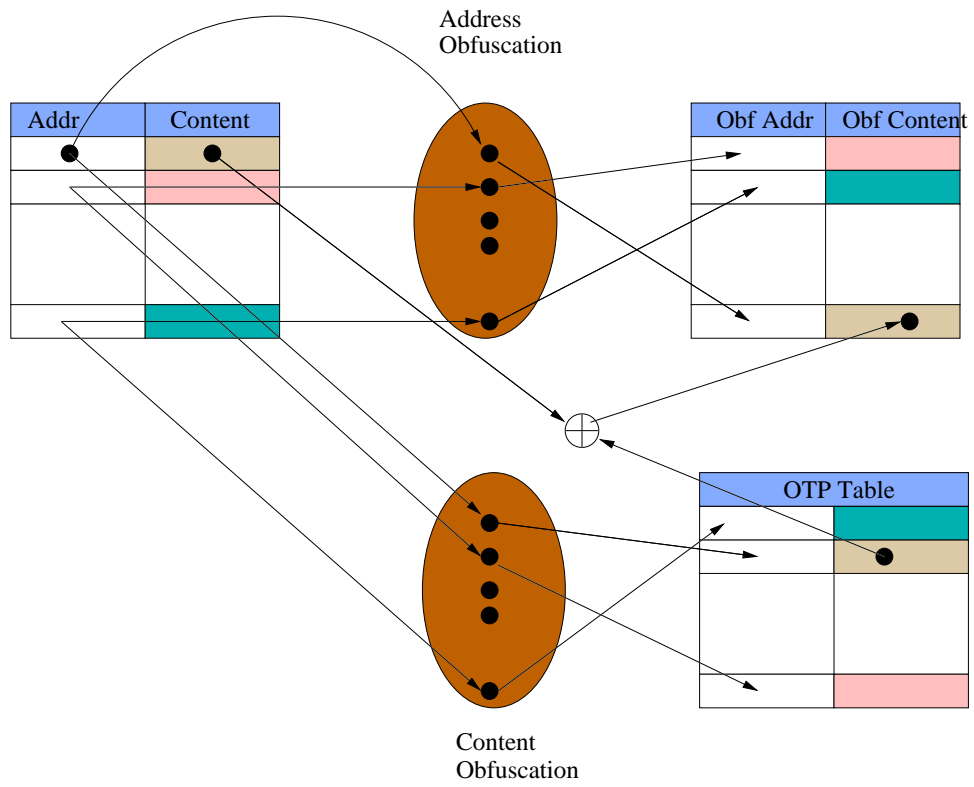


Figure 2.7 Page Obfuscation Function

additional properties on the target CPUs. For instance, it can restrict the distribution only to those machines which have a certain minimum *cache-block* size and *page* size, as both these parameters affect the strength of obfuscation. A suggested minimum for these parameters is  $64B$  and  $64KB$  respectively. The basis of static obfuscation is a page obfuscation function (*page\_obfuscate*) which takes an input page, an OTP page, and configurations for both address-sequence and content obfuscation functions. It produces an obfuscated output page. The outline of this algorithm is shown in Algorithm-1 and Figure 2.7. The algorithm for *static obfuscation* is shown in Algorithm-2.

---

Algorithm 2 Static Obfuscation Function: *stat\_obfuscate*

**Inputs**

$N_p \Leftarrow$  number of pages in the binary

$Page_{arr} \Leftarrow$  array of pages

**Function**

$p \Leftarrow$  temporary page

Generate random page of OTP ( $OTP_s$ )

**for**  $k = 0$  to  $N_p - 1$  **do**

**if**  $Page_{arr}[k]$  to be protected **then**

        Generate random  $S_{seq}$

        Generate random  $S_{cont}$

$Page_{arr}[k].p_{conf} = K_s\{S_{seq}, S_{cont}\}, HMAC$

$p = page\_obfuscate(S_{seq}, S_{cont}, OTP_s, Page_{arr}[k])$

$Page_{arr}[k] = p$

**end if**

**end for**

---

For every protected page the software vendor generates  $S_{seq}$ , the configuration for sequence obfuscation (corresponding to  $\pi_s$ ), and  $S_{cont}$ , the configuration for content obfuscation (corresponding to  $\pi_{c_s}$ ). It uses *page\_obfuscate* to obfuscate the page, and associate the configuration information with the page. This is shown in Algorithm-2. Even for pages which are not loaded, an obfuscation function could be associated. Note that *Arc3D* needs a standardized mechanism to garner these functions. This could be done by extending the standard binary format, like ELF, to hold the sections containing the configurations. The configurations have to be guarded, and hence need to be encrypted before being stored with the binary image. The software vendor has to generate a key,  $K_s$ , specific to this installation to support such encryption. All page level configurations,  $S_{seq}$  and  $S_{cont}$ , are encrypted with this  $K_s$ . And HMAC [27] of these

encrypted configurations is also generated. HMAC is a keyed hash which will allow *Arc3D* to detect any tampering of the encrypted configurations. Let  $P_{conf}$  represent the encrypted configurations and its HMAC and let  $S_{conf}$  represent the section containing  $P_{conf}$  of all the pages. The new binary format should carry encrypted configurations and its HMAC for every protected page. The page containing the cache-block OTPs also needs to be stored. This page is also encrypted with  $K_s$ . Its HMAC is computed as well. A new section  $S_{OTP}$  is created in the binary file and the encrypted OTP page and its HMAC are added to it.

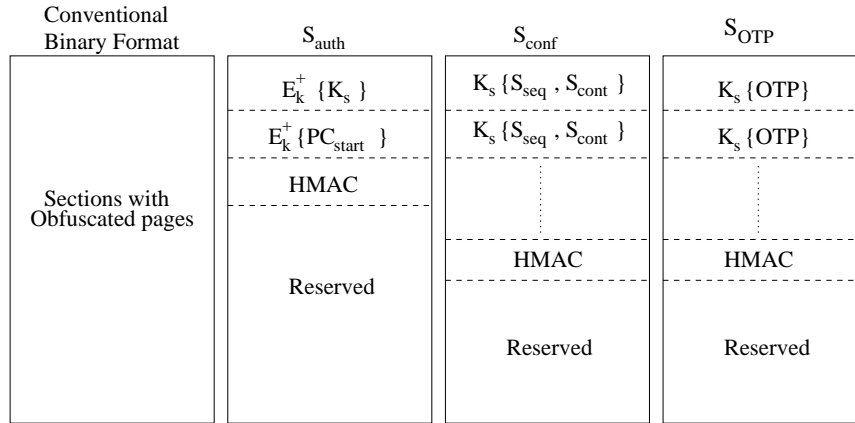


Figure 2.8 Extended Binary Format

---

### Algorithm 3 Software Distribution

- 1: Get  $E_k^+$  from CPU
  - 2: Contact CA and validate  $E_k^+$
  - 3: Generate  $K_s$
  - 4: Generate  $conf\_seq, conf\_cont$  for every page to be protected
  - 5: Generate  $OTP$  page
  - 6: Do *stat\_obfuscate*
  - 7: Generate  $S_{auth}$  and add it to binary file
  - 8: Generate  $S_{conf}$  and add it to binary file
  - 9: Generate  $S_{OTP}$  and add it to binary file
  - 10: Send the binary file to CPU
- 

In order for the CPU to be able to decrypt the program, it needs the key  $K_s$ . This is achieved by encrypting  $K_s$  with  $E_k^+$  and distributing it along with the software. Now only the CPU with the private key  $E_k^-$  can decrypt the distributed image to extract  $K_s$ . The entry point of the program also needs to be guarded. Several attacks are possible if the adversary could change the entry point. Hence, the entry

point is also encrypted with  $K_s$ . Once again we need to use HMAC to detect any tampering. Hence,  $S_{auth}$ , the authorization section, consists of  $E_k^+\{K_s, PC_{start}\}, HMAC$ . These extended sections are shown in Figure 2.8.

As we have argued earlier, the obfuscation process makes the software tamper resistant. In order to tamper the software in an undetectable and advantageous manner, the adversary must know the OTP. The probability of guessing an OTP is very small as we use  $64B$  length OTPs. Note that the encrypted contents of blocks from a different page do not leak any information about the OTPs. This is so because the  $i$ th OTP,  $OTP[b_i]$  is applied to different blocks  $\pi_S^P(i)$  and  $\pi_S^{P'}(i)$  in two different pages  $P$  and  $P'$ . Hence this form of obfuscation is at least as strong as the function guarding the configuration bits. The symmetric encryption with  $K_s$  guards the configuration bits. The symmetric key  $K_s$  can be of any arbitrary length. Its length will have little impact on the *Arc3D* performance(as shown later). The complete algorithm for software distribution step is shown in Algorithm-3.

## 2.5.2 Management of Protected Process

We now explain the OS use of the *Arc3D* APIs to manage a protected process. We will also show how seamlessly it can be integrated with the existing systems while providing the guarantees of *tamper-resistance* and *copy-protection*.

### 2.5.2.1 Starting a Protected Process

*Arc3D* has two execution modes, (1) protected and (2) normal, which are enforced without necessarily requiring the OS cooperation. When the OS creates a process corresponding to a protected program, it has to read the special sections containing  $S_{auth}$  and per-page configuration  $P_{conf}$ . *Arc3D* has an extended translation lookaside buffer ( $TLB_{xp}$ ) in order to load these per-page configurations. The decision whether to extend the page table entry (PTE) with these configuration is, OS and architecture dependent. We consider an architecture in which the TLB misses are handled by the software. Hence the OS can maintain these associations in a data structure different from PTEs. This will be efficient if very few protected processes (and hence protected pages) exist. This method is equally well applicable to a hardware managed TLB wherein all the PTEs have to follow the same structure.

The OS, before starting the process, has to update extended TLB with  $P_{conf}$ , for each protected page. Additionally, for every protected page, the OS has to set the protected mode bit  $P$ . This will be used by the architecture to decide whether to use the obfuscation function. Note that by entrusting the OS to set the  $P$  bit, we have not compromised any security. The OS does not gain any information or advantage by misrepresenting the  $P$  bit. For example, by misrepresenting a protected page as unprotected, the execution sequence will fail as both instructions and address sequences will appear to be corrupted. This is followed by the OS providing  $Arc3D$  with a pointer to  $S_{auth}$  and a pointer to  $S_{OTP}$ .

The OS executes *start\_prot\_process* to start the protected process execution. This causes  $Arc3D$  to transition to *protected* mode.  $Arc3D$  decrypts  $S_{auth}$  and checks its validity by generating its HMAC. If there is any mismatch between the computed and stored HMACs, it raises an exception and goes out of *protected* mode. If HMACs match, then  $Arc3D$  can start the process execution from  $PC_{start}$ . However, the address sequence generated at the address bus will expose the  $\pi_S$  function through one-to-one correspondence with the static binary image sequence. This compromises the static obfuscation. As explained in HIDE [59], the address sequence information suffices to reverse engineer the IP without even knowing the actual instructions. Hence,  $Arc3D$  performs one more level of obfuscation, called *dynamic obfuscation*, on protected pages to avoid these scenarios.

*Dynamic obfuscation* is very similar to the *static obfuscation*. It consists of two independent obfuscation functions per page, one to obfuscate the sequence of *cache-block* addresses, and the other to obfuscate the contents of *cache-blocks*. When *start\_prot\_process* is executed,  $Arc3D$  generates an OTP page ( $OTP_d$ ). This  $OTP_d$  needs to be stored in memory so that it can be reloaded at a later point after a context switch. We use the section  $S_{OTP}$  to store  $OTP_d$ .  $Arc3D$  has sufficient internal space to hold both  $OTP_s$  and  $OTP_d$  at the same time. It reads  $S_{OTP}$  and decrypts  $OTP_s$ , validates the HMAC, and then loads it into the obfuscation engine. It then encrypts  $OTP_d$  with  $K_s$  and generates its HMAC which is appended to  $S_{OTP}$ . We assume that the space for  $OTP_d$  in the section  $S_{OTP}$  has already been allocated at compile time.

$Arc3D$  then scans the TLB and validates  $P_{conf}$  for every protected page that has been loaded in the main memory. It then generates  $D_{seq}$  and  $D_{cont}$  configurations (corresponding to  $\pi_D$  and  $\pi_{c_d}$ ) for each one of those pages and appends them to their  $P_{conf}$ .  $TLB_{xp}$  which has been extended to hold  $P_{conf}$ , also has protected space per TLB entry which only  $Arc3D$  can access. This space will be used by  $Arc3D$  to store

the decrypted  $S_{seq}, S_{cont}, D_{seq}, D_{cont}$  configurations, so that decryption need not be done for every TLB access. *Arc3D* contains temporary buffer of twice the *page* size to perform the obfuscation. Hence it reads a complete page from RAM and applies *page\_obfuscation* and then stores it back in RAM. Algorithm for dynamic obfuscation is shown in Algorithm-4.

---

Algorithm 4 Dynamic Obfuscation Function: *dyn\_obfuscate*

**Inputs**

$N_{TLB} \leftarrow$  number of TLB entries

$p_i \leftarrow$  page to be obfuscated, read from RAM

$p_o \leftarrow$  obfuscated page

$OTP_d \leftarrow$  array of dynamic OTP

**Function**

**for**  $k = 0$  to  $N_{TLB} - 1$  **do**

**if**  $TLB[k].P$  is set **then**

**if**  $TLB[k].prot = NULL$  **then**

      Decrypt and validate  $P_{conf}$

**if**  $D_{seq}, D_{cont}$  exist **then**

$p_o = page\_unobfuscate(D_{seq}, D_{cont}, OTP_d, temp_i)$

$p_i = temp_o$

**end if**

      Generate new  $D_{seq}, D_{cont}$

      Append it to  $P_{conf}$

$TLB[k].prot = \{S_{seq}, S_{cont}, D_{seq}, D_{cont}\}$

      Read the page in  $p_i$

$p_o = page\_obfuscate(D_{seq}, D_{cont}, OTP_d, p_i)$

      Write back  $temp_o$

**end if**

**end if**

**end for**

---

The  $TLB[k].Prot$  structure is the protected section of TLB entry and is cleared every time a new TLB entry is written. Hence the function *dyn\_obfuscate* is invoked on every TLB miss. If the page has already been subjected to dynamic obfuscation, *Arc3D* first performs the inverse operation (deobfuscation). It then generates new obfuscation configurations to perform dynamic obfuscation. This causes the dynamic obfuscation functions to be very short lived, *i.e.*, changing on every page fault. It makes reverse engineering of  $\pi_D$  and  $C_D$  functions extremely unlikely. To ensure such a  $(\pi_D, C_D)$  refresh on every context switch,  $TLB[k].Prot$  is cleared for all the entries whenever *start\_prot\_process* is called or a protected process is

restored. A state register  $ST_i$  is allocated to the process and added to  $S_{auth}$ . The usage of this register is explained in Section 2.5.2.5. Availability of this register puts a limit on total number of protected processes active at any point in time in *Arc3D*. After the dynamic obfuscation is done, the process is started from  $PC_{start}$  as given by  $S_{auth}$ . The steps involved in *start\_prot\_process* are shown in Algorithm-5.

---

Algorithm 5 *start\_prot\_process*

- 1: Change to *protected* mode
  - 2: Read  $S_{auth}$  and validate
  - 3: Read  $S_{OTP}$  and validate
  - 4: Generate  $OTP_d$  and append to  $S_{OTP}$
  - 5: Clear  $TLB[i].prot$  for all  $i$
  - 6: Call *dyn\_obfuscate*
  - 7: Allocate  $ST_i$  to the process and add it to  $S_{auth}$
  - 8: Set PC to  $PC_{start}$
- 

### 2.5.2.2 Memory Access

Once a process is started it generates a sequence of instruction and data addresses. Like any high performance architecture, we assume separate TLBs, ITLB and DTLB, for instruction and data. Hence the loading process explained earlier occurs parallelly in both ITLB and DTLB. The TLB is the key component of the obfuscation unit. The obfuscation functions are applied only during virtual to physical memory mapping. The address generation procedure is outlined in Algorithm-6. Two stages of  $F_{obf}$  are in the computation path for the physical address. This makes TLB latency higher than the single cycle latency of a typical TLB access. Hence, L1 caches of both instruction and data are made *virtually tagged* and *virtually addressed* to reduce the performance impact due to TLB latency. The L1 cache tags are extended with a *protection* bit, which is accessible only to *Arc3D*. This bit is set whenever the cache line is filled with data from a protected page. The access to protected cache-blocks is restricted only in protected mode. In order to have efficient context switching mechanism we use a *write-through* L1 cache. Thus, at any point in time L2 and L1 are in synch.

TLB and L1 cache are accessed parallelly. TLB is read in two stages. The first stage reads the normal portion of TLB and the second stage reads the extended and protected portion of TLB. This way the second stage access can be direct mapped and hence could be energy-efficient. If L1 access is a hit,

---

 Algorithm 6  $TLB_{xp}$  Access Function:  $tlb_{xp\_access}$ 

```

v_page  $\leftarrow$  input virtual page address
v_block  $\leftarrow$  input virtual block address
p_addr  $\leftarrow$  output physical address
k  $\leftarrow$  TLB index of hit and page exists in RAM
if  $TLB[k].P$  is set then
  p_block =  $F_{obf}(D_{seq}, F_{obf}(S_{seq}, v\_block))$ 
else
  p_block = v_block
end if
p_addr =  $TLB[k].p\_page + p\_block$ 

```

---

then TLB access is stopped at  $stage_1$ . If L1 access is a miss, then TLB access proceeds as shown in the function  $tlb_{xp\_access}$ . In *Arc3D* L2 cache is *physically tagged and physically addressed*. Hence, no special protection is needed for the L2 cache. On an L2 cache access to an instruction in the middle of a cache-block, the relative intra-block sequence information is leaked to an observer adversary on the L1-L2 cache boundary. Given that for a 64B cache-block size, there are 16 instructions whose sequencing information is open to exposure. One way to lessen this vulnerability is to have L1 cache only issue L2 cache-block addresses. The cache-block offset can be retained by the L1 cache for later decoding. Hence the address traces visible at L1-L2 cache boundary will appear to be L2 cache-block address aligned. This would increase the latency but as we discuss later, this increase will not be very high. Once the data is received from the L2 cache or memory, it is *XOR*ed with both  $OTP_d$  and  $OTP_s$  to get the actual content in plaintext which is then stored in an L1 cache line.

### 2.5.2.3 Execution

*Arc3D* has a set of protected registers ( $REG_p$ ) to support protected process execution. This register set is accessible only in the protected mode. The protected process can use the normal registers to communicate with the OS and other unprotected applications. If two protected processes need to communicate in a *secure* way, then they have to use elaborate protocols to establish common obfuscation functions. Data sharing can also occur through a shared secret embedded into two applications by the software vendor in advance.

#### 2.5.2.4 Interrupt Handling

Only instructions from a protected page can be executed in protected mode. Hence any call to system services, such as dynamic linked libraries, requires a state change. Any interrupt causes *Arc3D* to go out of protected mode. Before transitioning to normal mode, *Arc3D* updates PC field in  $S_{auth}$  with the current PC. Thus a protected process context could be suspended in the background while the interrupt handler is running in the unprotected mode. When the interrupt handler is done, it can execute *ret\_prot\_process* to return to the protected process. *Arc3D* takes the PC from  $S_{auth}$  and restarts from that point. This allows for efficient interrupt handling. But from the interrupt handler, the OS could start other unprotected processes. This way *Arc3D* does not have any overhead in a context switch from protected to unprotected processes. But when the OS wants to load another protected process the current protected process' context must be saved.

#### 2.5.2.5 Saving and Restoring Protected Context

*Arc3D* exports *save\_prot\_process* API to save the current protected process context. This causes *Arc3D* to write  $K_s\{REG_p\} + HMAC$  and  $S_{auth}$  into the memory given by the OS. The OS when restoring the *protected* process, should provide pointers to these data structures through *restore\_prot\_process*. *Arc3D* can be enabled to detect *replay* attacks by including an association of time with the saved contexts. A set of OTP registers called state OTP registers are required within *Arc3D* for this purpose. These registers are the same size as  $K_s$ . The number of these registers depends on how many protected processes need to be supported simultaneously. The *start\_prot\_process* allocates a state OTP register  $ST_i$ . This association index  $ST_i$  is also stored within  $S_{auth}$ . Each instance of *save\_prot\_process* generates a state OTP value  $OTP[ST_i]$  which is stored in  $ST_i$ . The saved context is encrypted with the key given by the XOR of  $K_s$  and  $OTP[ST_i]$ . Symmetrically, an instantiation of *restore\_prot\_process* first garners  $ST_i$  and  $K_s$  from  $S_{auth}$ . Then the key  $OTP[ST_i] \oplus K_s$  is used to decrypt the restored context. This mechanism is very similar to the one used in all the earlier research such as ABYSS and XOM.

### 2.5.2.6 Supporting fork

In order to fork a protected process, the OS has to invoke *transfer\_prot\_process* API. This causes a new  $ST_i$  to be allocated to the forked child process. It then makes a copy of process context similar to *save\_prot\_process* handling. Thus the parent and the child processes could be differentiated by *Arc3D*. The OS has to make a copy of  $S_{OTP}$  for the child process.

### 2.5.2.7 Exiting a Protected Process

When a protected process finishes execution, the OS has to invoke *exit\_prot\_process* API to relinquish the  $ST_i$ . This is the only resource that limits the number of protected processes allowed in an *Arc3D* system. Hence *Arc3D* is susceptible to denial-of-service (DOS) kind of attacks.

### 2.5.2.8 Protected Cache

*Arc3D* has a protected direct mapped L2 cache of page size, *i.e.*, 64KB. This protected cache is used to obfuscate the second-order address sequences only for instructions, as temporal order doesn't have any meaning with respect to data. Whenever there is an IL1 miss in protected mode, *Arc3D* sends a request to  $L2_{prot}$ . Since  $L2_{prot}$  is on-chip, the access latency will be small. We assume it to be 1 cycle. If there is a miss in  $L2_{prot}$  then L2 is accessed.  $L2_{prot}$  is also invalidated whenever a protected process is started or restored.

## 2.6 Discussion

### 2.6.1 Assumptions

In this section we state and justify the underlying assumptions for *Arc3D*. The first and foremost of our assumptions is that every *Arc3D* processor has a unique identity (TPM's EK like identity). *Arc3D* device manufacturer can use various methodologies to embed the identity. Silicon Physical Random Functions (PUF) [25] have been proposed for this purpose. IBM's secure crypto-processor [22] provides a mechanism based on packaging for storing secrets within the processor environment. Xilinx [57] in its CPLD devices uses metal layers and dual access mechanisms to obfuscate the stored secrets.

The next issue is the extent of damage due to the exposure of *Arc3D* identity secret. If an adversary is able to gain access to the stored secret, then all the programs that were distributed for that particular instance of *Arc3D* could be decrypted. Once the program plaintext is obtained it can be executed in any *Arc3D* machine in unprotected mode. Hence the ability to protect the stored secrets within the architecture is of paramount importance in *Arc3D* design. However, the programs distributed to and encrypted for other *Arc3D* platforms are not compromised by the exposure of the secrets of a given platform.

### 2.6.2 Attack Scenarios

In this section we argue that *Arc3D* achieves our initial goals, namely, *copy-protection*, *tamper-resistance* and *IP-protection*. Several attacks causing information leak in various dimensions could be combined to achieve the adversary's goal. These attacks could be classified into two categories — attacks that target *Arc3D* to manipulate its control or reveal its secrets. If the adversary is successful in either getting the stored secret ( $E_k^-$ ) or in changing the control logic, the security assurances built upon *Arc3D* could be breached. But these type of attacks have to be based on *hardware*, as there are no software control handles into *Arc3D*. There are several possible hardware attacks, like Power Profile Analysis attacks, Electro magnetic signal attacks. The scope of this thesis is not to provide solutions to these attacks. Hence we assume that *Arc3D* is designed with resistance to these hardware attacks.

The second type of attacks are white-box attacks. Such an attack tries to modify the interfaces of *Arc3D* to the external world, to modify the control. The guarantees that are provided by *Arc3D* to the software in protected mode of execution are 3D obfuscation for protected pages based on the unique identity per CPU. Protected mode of execution guarantees that the control is not transferred to any unauthorized code (which is undetected). *Arc3D* will fault when an instruction from an unprotected page or from a page that was protected with different  $K_s$  is fetched in protected mode. This will prevent attacks of the buffer overflow kind. 3D obfuscation provides us both IP-protection and tamper-resistance. IP-protection is achieved because at every stage of its life, the binary image is made to look different, hence reducing the correlation based information leaks to the maximum extent possible.

Correlation based attacks are the ones where an adversary builds up information about the program behavior through repeated program executions. Such techniques [33] have been successfully used against

commercial secure microcontroller DS5002FP [20]. In *Arc3D* such attacks are prevented, as the dynamic obfuscation functions are chosen at random for every process run, which prevents incremental information gain.

Tampering could be performed by many means. But all of them have to modify the image of the process. Since every cache-block in every protected page potentially could have a different OTP, the probability that the adversary could insert a valid content is extremely small. Applications can obfuscate new pages that are created at run-time by designating them as protected. Applications can further maintain some form of Message Digest for sensitive data, because obfuscation only makes it harder to make any educated guess, while random modification of data is still possible. In the case of instructions, the probability that a random guess would form a *valid* instruction at a valid program point is extremely small.

Another form of tampering - splicing attack - uses valid cipher texts from different locations. This attack is not likely to succeed because every *cache-block* in every *page* has a unique OTP and every *page* has a unique address obfuscation function. This makes it hard for the adversary to find two *cache-blocks* with the same OTP. Another common attack is *replay attack*, where valid cipher text of a different instance of the same application is presented (replayed) to the CPU. As we discussed earlier, this attack is prevented by XORing  $K_s$  with a randomly generated OTP which is kept in the *Arc3D* state. This value is used as a key to encrypt the *protected* process' context. Thus when restoring a protected context, *Arc3D* makes sure that both  $S_{auth}$  and saved context are from the same run.

When the adversary knows the internals of the underlying architecture, another form of attack is possible. This form of attack denies resources that are essential for the functioning of the underlying architecture. For example, XOM maintains a session table and has to store a *mutating register* value per session-id. This mutating register is used to prevent any replay attacks. This kind of architecture has an inherent limitation on the number of processes it can support, *i.e.*, the scalability issue. Thus an attacker could exhaust these resources and make the architecture non-functional. This kind of attack is possible in *Arc3D* as well on the state OTP register file. We could let the *context-save* and *context-restore* be embedded in the storage root of trust in a TPM like model. Such a model will allow *Arc3D* to perform in a stateless fashion which can prevent the resource exhaustion attacks.

## 2.7 Performance Analysis

Since *Arc3D* seamlessly fits into the existing memory hierarchy as an extended TLB, the latency caused by *Arc3D* should be minimal. We used SimpleScalar [10] Alpha simulator with memory hierarchy as shown in Figure 2.2 to do the performance simulation. We did two sets of simulations with different latency parameters, Alpha 21264 and Intel XSCALE 80200 as shown in Table 2.2.

Table 2.2 Memory Hierarchy Simulation Parameters

Param	Alpha 21264 [19]	Intel XSCALE 80200 [58]
L1	64KB, 2 way, 64B, 3 cyc	32KB, 32-way, 32B, 3 cyc
ITLB/ DTLB	128 fully associative, 1 cyc	32 fully associative, 1 cyc
L2	1MB, 1 way, 16 cyc	256K, 8 way, 8 cyc
Memory	Lat 130 cyc, 4 bytes/cyc	Lat 32 cyc, 4 bytes/6 cyc
Peak B/w	7.1 GB/s	800 MB/s
Page sz	64KB	64KB

Three latencies are added by *Arc3D*, namely, extended TLB access, increased access time to L2 because of sending only block address to L2, and latency to read the pages and obfuscate them on every TLB miss. The first component gets absorbed in L1 cache access latency for both the systems, assuming that the extended TLB access increases the TLB access latency by 2 cycles. The major component is the reading time of *page* and writing it back to the memory. Since obfuscation is just an XOR operation, we can assume it takes one cycle. These facts along with the assumption that these pages are transferred in and out of *Arc3D* at the peak memory bandwidth, lead to a latency increase of 12,000 cycles in the case of Alpha-2164 and 96,000 cycles in the case of XSCALE. The simulation was run with Spec2000 [48] benchmarks for 2 Billion instructions by fast-forwarding the first 500 million instructions.

Table 2.3 shows that the performance impact on XSCALE 80200 memory hierarchy with higher number of TLB misses is greater than the impact on Alpha 21264 memory hierarchy. On Alpha 21264 the performance impact is less than 1% for most of the benchmarks.

Table 2.3 Simulation Results

XSCALE 80200					
Bench	IL1 Missrate	DL1 Missrate	ITLB Misses	DTLB Misses	%CPI Increase
bzip	0.0000	0.0225	2	256408	479
eon	0.0000	0.0020	10	12	0.145
gcc	0.0037	0.0510	28	110636	509
twolf	0.0000	0.0728	7	31	0.128
crafty	0.0009	0.0051	6	15627	73.4
gzip	0.0000	0.0231	3	1906	10.6
parser	0.0000	0.0354	5	50663	245
Alpha 21264					
Bench	IL1 Missrate	DL1 Missrate	ITLB Misses	DTLB Misses	%CPI Increase
bzip	0.0000	0.0185	2	113	0.12
eon	0.0000	0.0008	10	12	0.02
gcc	0.0019	0.0272	29	1804	0.97
twolf	0.0000	0.0508	7	31	0.01
crafty	0.0002	0.0123	6	33	0.02
gzip	0.0000	0.0125	3	1906	1.12
parser	0.0000	0.0210	5	1121	0.74
vpr	0.0000	0.0444	5	51	0.05

## 2.8 Conclusion

Software obfuscation is a key technology in IP-protection. However, software only solutions (such as compiler transformations of control flow or insertion of redundant basic blocks or data structure transformations) often do not have robustness of crypto methods. Complete control flow obfuscation methods such as Cloakware [12] have the limitation that they cannot hide the correct control flow information from the prying eyes of the OS/end user. An additional weakness in these schemes is that observation of repeated dynamic execution often gives away the obfuscation secrets (such as control flow ordering or data structure sequencing).

We propose a minimal architecture, *Arc3D*, to support efficient obfuscation of both static binary file system image and dynamic execution traces. This obfuscation covers three aspects: address sequences, contents, and second-order address sequences (patterns in address sequences exercised by the first level of loops). We describe the obfuscation algorithm and schema, its hardware needs, and their performance impact. We also discuss the robustness provided by the proposed obfuscation schema.

A reliable method of distributing obfuscation keys is needed in our system. The same method can be used for safe and authenticated software distribution to provide copy-protection. A robust obfuscation also prevents tampering by rejecting a tampered instruction at an adversary desired program point with an extremely high probability. Hence obfuscation and derivative tamper-resistance provide IP-protection. Consequently, *Arc3D* offers complete architecture support for copy-protection and IP-protection, the two key ingredients of software DRM.

## **CHAPTER 3. TIVA: TRUSTED INTEGRITY VERIFICATION ARCHITECTURE**

We are moving towards the era of pervasive computing. The embedded computing devices are everywhere and they need to interact in many insecure ways. Verifying the integrity of the software running on these devices in such a scenario is an interesting and difficult problem. The problem is simplified if the verifying entity has access to the original binary image. However, the verifier itself may not be trusted with the intellectual property built into the software. Hence an acceptable and practical solution would not reveal the intellectual property (IP) of the verified software, and yet must verify its integrity. We propose one such novel solution, TIVA, in this chapter.

### **3.1 Introduction**

We are entering the era of pervasive computing where embedded devices have penetrated most spheres of human activity. These embedded devices carry a wide range of data ranging from sensitive personal information to military confidential information. Moreover, these devices need to interact frequently with the insecure world. Hence it is imperative to check frequently whether any malicious tampering of the software on these devices has occurred.

The different scenarios where such verification is beneficial, for example, are as follows.

- The field officer would like to ensure that her GPS has not been tampered with before entering the enemy territory. Note that the tampering adversary here is the GPS device. The military needs to distribute the binary image of the GPS software to the verifier so that the field officer can use the verifier to ascertain the integrity of the GPS software. The military, however, would be increasing the risk of compromising the IP of the GPS software by distributing the binary image to the verifier. Note that the IP adversary is the verifier (and not the device, which is a tampering adversary). The

problem then is devising verification engine (verifier) architecture to minimize the risk of exposing the IP of the distributed GPS software.

- An executive would like to ensure that the software and/or data on her PDA has not been tampered with. She could have a verifier installed on her laptop to verify the PDA. There exists a conflict of interest between the software vendor and the PDA user. PDA user (or the laptop version of the verifier) requires the binary image of the PDA software for verification. The software vendor may be at the risk of compromising the IP of her software by distributing it to the PDA owner. Thus the verification architecture should safeguard both the party's interests.
- An organization would like to ensure that their routers were not tampered with. This case is pretty similar to the earlier one except that the verification would be performed remotely. The verification architecture should be robust enough to support the remote verification of the systems.

All these scenarios demand IP protection in addition to the mere verification of the software. The existing solutions like *SWATT*([2]) and *Genuity*([45]) do not address the concern of IP protection and are very restricted to a certain class of devices hence not generally applicable.

Reverse engineering the low level code into a high level programming language is usually the first step in determining the embedded IP of a software. Such reverse engineering can lead to software piracy. Reverse engineering requires disassembling and decompilation of the instruction sequence. Static obfuscation techniques address this issue by hiding the instantiated instruction sequence. These obfuscation techniques embed the correction points for the control flow (the correct instruction sequence) in the image itself. Such instruction sequence obfuscation, however, applies only to the static program image. The instantiated instruction sequence is exposed during an execution.

In the case of verification model, the verifier needs the binary image for verification purpose only and not for execution purpose. In other words, an instantiated control flow path order is not important to the verifier. The verifier mostly needs only the memory address-content correspondence. Thus any obfuscation technique which modifies the static sequence of instructions need not embed the image with correction points. Such an obfuscated image becomes extremely hard to reverse engineer without the execution address sequence. In TIVA we use a permutation function to generate such an obfuscated image

in order to provide the IP protection.

TIVA uses challenge-response protocol between the verifier and the embedded device. In order to keep the tampering adversary, the device, honest in its responses, the challenge has to be different (unique) for each verification. TIVA uses a unique permutation function for each verification to calculate a unique checksum or hash. The novelty of TIVA lies in the fact that it can achieve both IP protection and challenge-uniqueness through the use of a permutation function. TIVA uses a trusted hardware element in the embedded device to achieve this. But this trusted hardware is different from TCG or secure processors as it has very minimal hardware overhead.

The main contributions of TIVA are

- identifying the need for IP protection for any practical integrity verification model for embedded devices
- providing both IP protection and challenge-uniqueness to every verification instantiation through permutation functions
- a reconfigurable circuit to achieve these permutation functions

The rest of this chapter is organized as follows. Section 3.2 describes the problem and the assumptions under which the solution is valid. Section 3.3 explains the proposed solution. In Section 3.5 we explain the overall verification architecture. Section 3.6 discusses strengths and weaknesses of our proposed verification architecture. Section 3.8 concludes this chapter.

## 3.2 The Problem

Integrity verification allows the verifier to assert that the binary image, which includes both code as well as static data, is as expected. Let  $\mathcal{E}$  be the device whose binary image needs to be verified,  $\mathcal{V}$  be the entity which would like verify the integrity and  $\mathcal{D}$  be the entity which distributes the software image  $I$ . The interactions between the entities are as follows. Recall that  $\mathcal{E}$  is the tampering adversary for the verification. However  $\mathcal{V}$  is the DRM adversary against whom we need to protect the software IP. Note that  $\mathcal{V}$  is a logical entity that can be physically realized either as a hardware or software unit separate from  $\mathcal{E}$  or it could be physically integrated as a software process or hardware unit within  $\mathcal{E}$ . In the later case, the

hardware version of  $\mathcal{V}$  would have to be secured against observation and tampering from  $\mathcal{E}$ . The software process version would have to be obfuscated and hidden within  $\mathcal{E}$  along the lines of software watermarking [17] with unique secret handles for instantiating the verification and for observing the outcome.

- *Distribution:* Software vendor  $\mathcal{D}$  distributes the image  $I$  to verifier  $\mathcal{V}$  to verify the integrity of the corresponding software in the device  $\mathcal{E}$ .
- *IP Protection:*  $\mathcal{D}$  trusts the device  $\mathcal{E}$  to have sufficient protection mechanism to protect the IP of image  $I$ . Note that  $\mathcal{E}$  is protecting the IP of  $I$  against possible reverse engineering by  $\mathcal{V}$ . However, a direct distribution of  $I$  to verifier  $\mathcal{V}$  by software vendor  $\mathcal{D}$  increases the risk of IP compromise.  $\mathcal{V}$  could have simulation/emulation environment or use other mechanisms to reverse-engineer  $I$ . To avoid such a scenario  $\mathcal{D}$  would like to ensure that IP of the binary image  $I$  is protected despite its distribution to  $\mathcal{V}$ .
- *Verification:*  $\mathcal{V}$  would like to verify the integrity of the binary image  $I$  resident in the device  $\mathcal{E}$ . The verification process should be challenge-response based, i.e. the verifier  $\mathcal{V}$  should be able to generate a challenge at random, and based on the response from  $\mathcal{E}$  should be able to assert the integrity of the image  $I$ . The verification process should be robust enough so that it is able to detect replay and spoofing attacks.

The problem boils down to  $\mathcal{V}$  verifying the image of  $\mathcal{E}$  with respect to  $I$  without revealing its IP under the condition that  $\mathcal{E}$  is not tampered with in hardware. The binary image refers to both the code as well as static data.

### 3.3 The Solution

The three dimensions of the problem as explained in Section 3.2 are *distribution*, *IP protection*, and *verification*. We first present the solution to the problems of verification and IP protection. The distribution problem arises out of this solution.

A straightforward solution to the problem of verification would be to distribute the binary image  $I$  to the verifier  $\mathcal{V}$ . Hence the verifier can read contents of the  $\mathcal{E}$  and compare it against the received image. But there are several problems with this simple and seemingly perfect scheme.

First of all the requirement of IP protection is violated by this scheme, as the verifier  $\mathcal{V}$  could very well be an attacker who would like to reverse engineer the IP of the image  $I$ . Another problem with this scheme is that it is highly inefficient. It will take time proportional to  $N * c$ , where  $c$  is the number of cycles required to read the memory content from the device  $\mathcal{E}$  and  $N$  is the size of the memory.

Earlier solutions like, *Genuinity*[45] and *SWATT* [2], addressed this problem by having a verification module in the device  $\mathcal{E}$ . This verification module receives the challenge and provides a response to the verifier  $\mathcal{V}$ . This verification logic is critically dependent on the following two dimensions.

1. binary image residing in the memory,  $I$ .
2. time to perform the verification,  $\mathcal{T}$ .

In such a verification module architecture, one solution would be to distribute the hash of the binary image to the verifier  $\mathcal{V}$ , and to ask the verification logic in the device  $\mathcal{E}$  to generate the hash as well, followed by a comparison of the two hashes. Any modifications in the binary image  $I$  will modify the hash and any modification to the verification logic to misrepresent the hash itself will result in a perceptible change to  $\mathcal{T}$ . Since only the hash is available to  $\mathcal{V}$ , no binary image  $I$  is provided, the IP protection problem is moot. But the drawback of such an approach is that the hash used in the verification is fixed. Any malicious software running in  $\mathcal{E}$  could spoof the verifier by responding with the fixed hash without having to recompute. The time to perform verification could be easily spoofed by the use of timers.

An alternative would be to request the verification module in the device  $\mathcal{E}$  to compute the hash of a variable subset of the image  $I$ . Since verifier  $\mathcal{V}$  can specify the subset at random the response to every challenge has to be uniquely calculated to thwart the replay attack. Similar method is used in AOL [1] and AIM [44]. In this schema though the verifier  $\mathcal{V}$  needs to be able to calculate the correct hash for more or less any subset of  $I$  (every challenge). It requires the entire binary image  $I$  for this ability. But this violates the IP protection requirement of our problem statement.

Yet another solution is to use keyed hash. The verifier  $\mathcal{V}$  can generate a random key and request the device  $\mathcal{E}$  to generate the hash for that key. This could also avoid the replay attack since the hash value depends on the key and the key is generated at random by the verifier  $\mathcal{V}$ . But this model also violates the IP protection requirement since the verifier  $\mathcal{V}$  requires the image  $I$  in order to calculate the hash for a randomly generated key. Another drawback especially applicable to a software based remote verifier

is the ease of mimicking the device behavior. An impostor device  $\mathcal{E}'$  could replace  $\mathcal{E}$  such that both are behaviorally equivalent (say a malicious router). Moreover,  $\mathcal{E}'$  could be computationally much more powerful than  $\mathcal{E}$ , able to easily calculate the hash within  $\mathcal{T}$  from the unmodified original image. In reality, though,  $\mathcal{E}'$  could be executing a modified malicious image. Since there is no shared secret between the verifier  $\mathcal{V}$  and the device  $\mathcal{E}$ , any impostor could generate the correct hash, since the hash algorithm, key and the image are all known to the impostor. This idea was also used by Umesh et al. [47] to attack *Genuinity* [45].

Thus the solution to the verification problem is to find an *irreversible* hash or checksum function which generates a unique hash  $\mathcal{H}$  for every verification. This function should be such that within the given time  $\mathcal{T}$  the only way to generate  $\mathcal{H}$  is to execute the given verification function. Also this function should share a secret with the verifier. Thus if  $\mathcal{E}$  returns the required  $\mathcal{H}$  within the specified time  $\mathcal{T}$  then it verifies the integrity of the device  $\mathcal{E}$  as well as the image in  $\mathcal{E}$ . Thus heart of the solution is in defining the irreversible hash function  $\mathcal{F}$  which generates  $\mathcal{H}$ . This  $\mathcal{F}$  and  $\mathcal{T}$  together constitute the signature of  $\mathcal{E}$  which is verified against the precomputed values by  $\mathcal{V}$ . This is the core of our proposed approach to integrity verification.

The required and desirable properties of the irreversible hash generation function  $\mathcal{F}$  are as follows.

1. It should be very fast and efficient. Hence any change in  $\mathcal{F}$  or its simulated/emulated version should result in a perceptible and observable change in the response time  $\mathcal{T}$ .
2. It should depend on the image  $I$  as well as on the challenge from the verifier  $\mathcal{V}$ . Thus for two distinct challenges, it should generate distinct hash values.

---

Algorithm 7 Irreversible Hash Function (Pseudocode)

```

for  $l = 0$  to  $N - 1$  do
     $hash = hash + (MEM[l] \oplus \pi(l))$ 
end for

```

---

Algorithm-1 shows such an irreversible hash function. This hash function calculates the checksum of the image  $I$  exor-ed with permutation function  $\pi$ .  $MEM[l]$  refers to memory contents of the image at location  $l$ .  $\pi$  refers to the permutation function which takes in a value from  $0 \cdots N - 1$  and returns a value from  $0 \cdots N - 1$ . There are  $N!$  possible distinct permutation functions. Verifier  $\mathcal{V}$  chooses a particular

permutation function through the challenge. Device  $\mathcal{E}$  should use that specific permutation function while calculating the checksum.

The notable characteristic of the hash function shown in Algorithm-1 is that it uses the permutation function  $\pi$  to create the dependency between checksum calculation and verifier's challenge. In contrast, SWATT [2] used pseudo-random generator and Genuinity [2] used architectural side-effects to introduce such dependency. The main reason behind our choice of permutation function is the additional capability of IP protection offered by these permutation functions.

Reverse engineering is the first step in determining IP of the software. In order to reverse engineer the control flow graph (CFG) of the image has to be reconstituted. This is done by disassembling and decompilation of the binary image. Various static obfuscation techniques ([35],[43],[16]) try to achieve IP protection by either obscuring the disassembling stage or decompilation stage. But these techniques are limited by the fact that the statically obfuscated image should retain the same CFG as its original.

The degree of obfuscation required in our problem is significantly weaker. The verifier  $\mathcal{V}$  needs the image only for verification or to establish address by address correspondence of the contents of  $\mathcal{V}$ 's and  $\mathcal{E}$ 's images. The binary image held by  $\mathcal{V}$  is not executed. This weakens the obfuscation constraints as follows. Any static obfuscation applied to the binary image  $I$  distributed to  $\mathcal{V}$  need not retain the original CFG. Any permutation of the sequence of the bytes in the binary image  $I$  would obfuscate the CFG, in turn making the reverse engineering extremely difficult. Thus obfuscated image  $I_{obf}$ , which is a permuted version of the image  $I$  could be distributed to the verifier  $\mathcal{V}$  without compromising its IP. Section 3.6.1 discusses in detail the strength of obfuscation function realized by permutation.

Our solution to the integrity verification problem which combines the permutation function to generate  $I_{obf}$  and the permutation function to generate hash to form a unified solution is as follows.

1. For every  $(\mathcal{V}, \mathcal{E})$ ,  $\mathcal{D}$  generates a permutation function  $\pi_d$  and gives  $(\pi_d(I), \mathcal{T})$  to  $\mathcal{V}$ .
2.  $\mathcal{D}$  secretly embeds  $\pi_d$  in  $\mathcal{E}$ .
3. For every verification,  $\mathcal{V}$  generates  $\pi_v$  and finds  $\mathcal{F}(\pi_v(\pi_d(I)))$ . It then gives  $\pi_v$  as a challenge to  $\mathcal{E}$ .
4.  $\mathcal{E}$  generates hash using  $\pi_v$  and  $\pi_d$  and reports it back to  $\mathcal{V}$ .
5.  $\mathcal{V}$  measures the response time  $\mathcal{T}$ .

6.  $\mathcal{V}$  can verify this *signature* with the precomputed one.

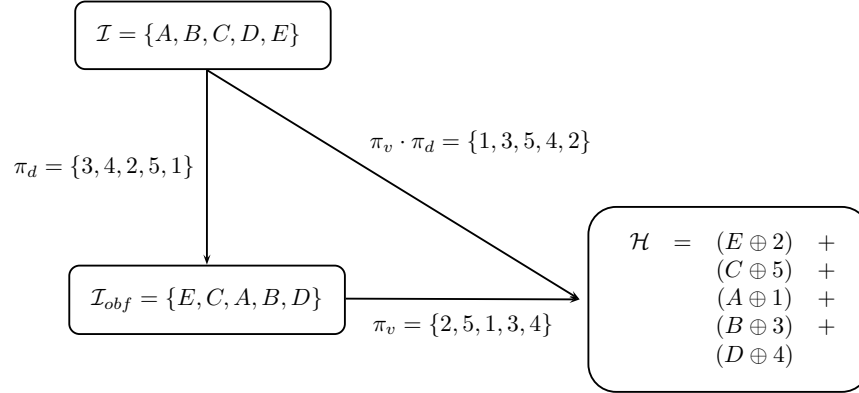


Figure 3.1 An Example *Hash* or *Checksum* Function  $\mathcal{F}$

Figure 3.1 shows an example calculation of checksum by both  $\mathcal{V}$  and  $\mathcal{E}$ . In this figure obfuscated image  $I_{obf}$  is generated as follows. Let  $M_{obf}$  be the memory content of  $I_{obf}$  and  $M$  be the memory content of image  $I$ . Then  $M_{obf}[\pi_d(i)] = M[i]$  for every  $i$  from 1 to  $N$ , where  $N$  is the size of the image. Note that image  $I$  is not necessarily limited to only instructions. The presence of static data could also obscure the disassembly which makes reconstruction of CFG more difficult. In this figure, verifier  $\mathcal{V}$  has the obfuscated image  $I_{obf}$  and device  $\mathcal{E}$  has the actual image  $I$ .  $\mathcal{V}$  generates  $\pi_v$  and calculates hash  $\mathcal{H}$  using Algorithm-1. Device  $\mathcal{E}$  uses the composite permutation function  $\pi_v(\pi_d)$  and the actual image  $I$  to calculate the same hash  $\mathcal{H}$ .

A permutation function  $\pi$  with  $N$  values is  $N!$  strong, which is slightly higher than  $2^N$  by Sterling's approximation of a factorial. Hence by choosing sufficiently large  $N$  we can reduce the probability of success through a *brute-force* attack. By choosing a different permutation function for every verification we avoid the *replay* attack. Attack by impostor is avoided as  $\mathcal{V}$  and  $\mathcal{E}$  share the permutation function  $\pi_d$  as the secret. Hence the impostor needs to know  $\pi_d$  to generate  $\mathcal{H}$ . We have assumed that  $\mathcal{E}$  is protected enough not to reveal its stored secrets.

The distribution of the software image now involves four operations, namely, distributing image  $I$  to the device  $\mathcal{E}$ , generating the permutation function  $\pi_d$ , generating the obfuscated image  $I_{obf}$  and distributing it to the verifier  $\mathcal{V}$ . Various existing solutions are applicable to this problem. In the case of embedded

devices it is most likely that the device vendor distributes the image as well. Hence the device vendor can maintain the association of  $\pi_d$  with the device's unique ID. Whenever the device is purchased or obtained by the verifier the vendor can generate the obfuscated image using  $\pi_d$  and distribute it with the device. Whenever the device needs to be updated with newer version of the image  $I$  the device vendor has to generate the corresponding  $I_{obf}$  and distribute it to the verifier  $\mathcal{V}$ . We use the reconfigurable permutation unit (RPU) described in Section 2.4.3 to achieve this purpose. This reconfigurable permutation unit (RPU) needs to be embedded into  $\mathcal{E}$ .

### 3.4 Area and Delay Estimation of RPU

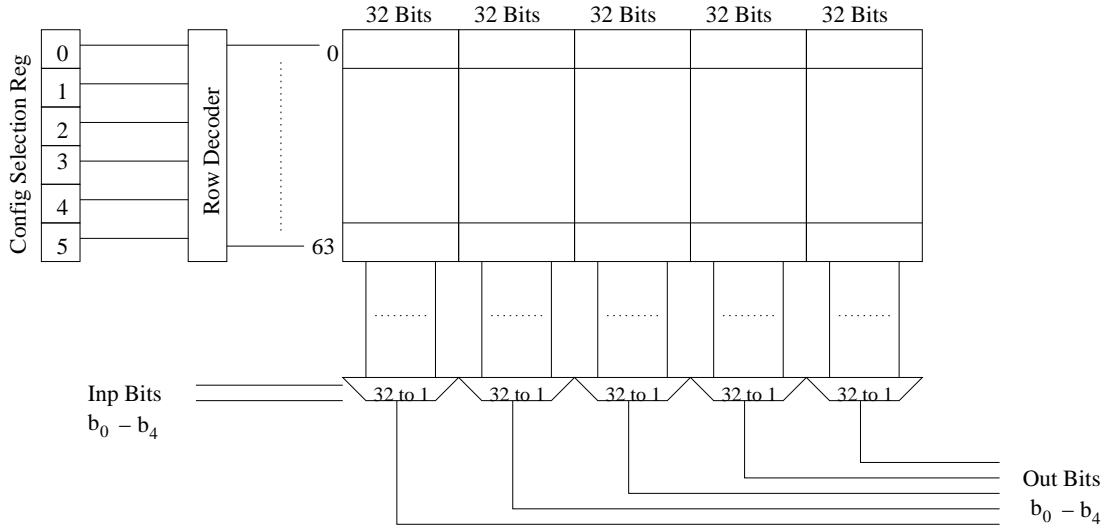


Figure 3.2 A Typical Schema for 5x5-LUT

Since we intend to use RPU in the embedded devices it should be both area and delay efficient. Figure 2.4 shows that RPU has 6  $5 \times 5$  - LUTs and 3 shifters. Each of these  $5 \times 5$  - LUTs takes 6 configuration selection bits and 5 input bits. Each LUT can be visualized as having a direct mapped cache with 64 sets and 32 bit cache line. Each cache line stores the configuration bits and one of which is chosen by the 6 configuration selection bits. One of these 32 configuration bits is chosen by the 5 input bits. Thus an LUT has a 256B direct mapped cache and a 32-to-1 multiplexer. Since all the 5 LUT use the same configuration selection bits we can group all these direct mapped caches and make it a single

direct mapped cache with 64 sets and 20 byte cache lines. Figure 3.2 shows such a schema.

Table 3.1 Area Estimate of RPU

Technology <i>nm</i>	Area <i>mm</i> <sup>2</sup>
180	1.4526
130	0.7578
70	0.2196

Thus RPU has 6 1.25KB direct mapped caches. Since configuration selection bits will be preloaded, the delay incurred in accessing these caches would not have any impact on the access time of RPU. We used CACTI[49] to estimate the area requirement of RPU and Table 3.1 lists the area estimate for various process technologies. The other components of RPU are shifters and multiplexers. The shifters could be realized through 2-to-1 multiplexers. Since more than 99% of the transistors of RPU are contributed by the caches the area estimate of RPU could be equated to the area estimate of the caches.

To estimate the access time of RPU we should find the components which contribute to the access time. Since the configuration selection register will be preloaded the configuration bits will be available to the multiplexers. The access time can be given as,

$$T_{RPU} = 3 \times T_{32-to-1 MUX} + 2 \times T_{2-to-1 MUX}$$

We used HSPICE[28] to perform the delay estimation. We used pass transistor logic with appropriate drivers to design the multiplexers as they are area efficient. In order to optimize the delay of a 32-to-1 multiplexer we designed it as a 3-level multiplexer with first two levels being 4-to-1 multiplexers and the last one being 2-to-1 multiplexer. We used TSMC[54] and BPTM[9] models for the simulation. The results of the simulation are listed in Table 3.2. We will use these delay estimates while estimating the latency of this functional unit in the following section.

Table 3.2 Delay Estimate of RPU

Technology $nm$	Model	$V_{cc}$ V	$V_{th}$ V	$T_{32-to-1}$ ps	$T_{2-to-1}$ ps	$T_{RPU}$ ns
180	TSMC[54]	1.80	0.46	369	70	1.247
180	TSMC[54]	1.30	0.28	410	80	1.390
180	TSMC[54]	1.55	0.28	340	60	1.140
70	BPTM[9]	0.90	0.20	220	60	0.780

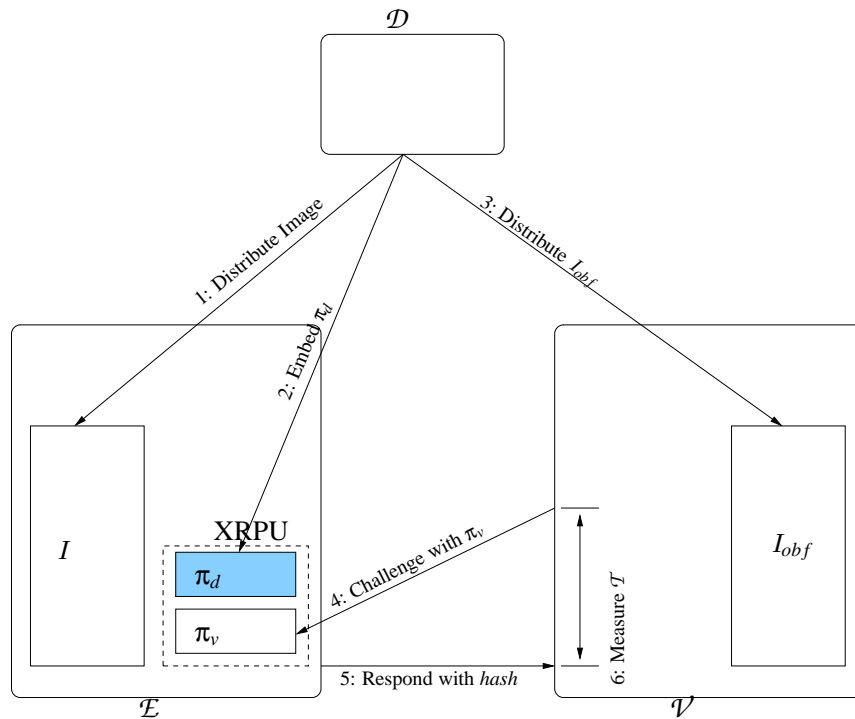


Figure 3.3 Integrity Verification Architecture

### 3.5 Integrity Verification Architecture

In Section 3.3 we outlined our basic solution for embedded device verification. In this section, we explain TIVA in more details. TIVA uses RPU, the hash function  $\mathcal{F}$  and the response time  $\mathcal{T}$  to provide the solution to the integrity verification problem.

#### 3.5.1 XRPU

As explained in Section 3.3, the IP of image  $I$  is protected through  $\pi_d$ . In TIVA this is achieved by embedding this secret in the device  $\mathcal{E}$ . Hence  $\mathcal{E}$  should have a protected hardware where this secret could be stored. From Section 2.4.3, we know that RPU has space to store all the 64 possible configurations for each LUT. Since  $\pi_d$  chooses only one of these configurations we do not need to store all of them. Thus  $\mathcal{E}$  should have a special RPU $_{\pi_d}$  which stores only the chosen configuration bits which amounts to 120 bytes for the LUTs and 3 bits for the exchangers.

$\mathcal{E}$  should contain a second RPU $_{\pi_v}$  which is a generic one as explained in Section 2.4.3. This RPU is loaded with the configuration selection bits generated by  $\mathcal{V}$ . Since we want to protect the function  $\pi_d$ , we do not allow the input/output relation of RPU $_{\pi_d}$  to be visible. If  $\pi_d$  is allowed to be observed then  $I_{obf}$  could be de-obfuscated resulting in loss of its IP. Thus we create a single composite function unit XRPU, eXtended RPU, which contains both RPU $_{\pi_d}$  and RPU $_{\pi_v}$ . It takes start address and configuration selection as input and produces the *hash* as the output. This XRPU generates all 1024 addresses sequentially from the start address and computes  $hash = hash + \{MEM[addr] \oplus \pi_d(\pi_v(addr))\}$ . This could be implemented as microcode or implemented in hardware. Since  $\pi_v$  is public its permutation function is known. Hence given  $addr$  and  $MEM[addr] \oplus \pi_d(\pi_v(addr))$  it is easy to derive  $\pi_d$ . Thus XRPU only provides *hash* as the output from which  $\pi_d$  cannot be obtained as it is an irreversible function.

#### 3.5.2 Verification

As is the case with any encryption function, the algorithm of RPU is public. The secret is the *configurations bits*. Thus  $\mathcal{V}$  could be provided with a simulated version of RPU's algorithm or it could have a special application-specific hardware unit. To verify the authenticity of the image,  $\mathcal{V}$  generates the configuration bits for  $\pi_v$  randomly and computes the checksum as  $sum = sum + (MEM[i] \oplus \pi_v(i))$ . It then

---

```

li    0,0    ; R0 counter
li    5,0    ; R5 LS word of checksum
li    6,0    ; R6 MS word of checksum
lwz   1,st   ; R1 starting address
L1:  add  1,0,1 ; add counter to address
      xrpu 3,0   ; R3 = XRPU(R0)
      lwz  2,0(1) ; load the content in R2
      xor  3,2,3 ; R3 = R3 xor R2
      srawi 4,3,31 ; R4 = sign bit of R3
      addc 5,5,3 ; R5 = R5 + R3
      adde 6,6,4 ; R6 = R6 + R4 + Carry
      addi 0,0,1 ; R0 = R0 + 1
      cmpwi 0,0,1024; is R0 < 1024
      lt   L1    ; loop back

```

---

Figure 3.4 An Example PPC Micro-Code Implementation of  $\mathcal{F}$

sends this  $\pi_v$  as a challenge to  $\mathcal{E}$  and measures the time of verification (response time).

Since XRPU is a hardware unit, the verification function  $\mathcal{F}$ , which we assume to be a microcode, could be very fast. As an example, in PPC the execution of one iteration of loop body for this function takes only 10 cycles assuming XRPU takes 2 cycles per operation. Example pseudocode is shown in Figure 3.4. This is very fast and efficient. Any small modification in the verification code results in perceptible change in the time  $\mathcal{T}$  of the verification process. Thus from the checksum and  $\mathcal{T}$ ,  $\mathcal{V}$  can establish the integrity of the binary image in  $\mathcal{E}$ . Figure 3.3 explains various steps involved in the verification architecture.

### 3.5.3 Overhead Estimation

Since  $\text{RPU}_{\pi_d}$  stores only one set of configuration bits area of XRPU  $\approx$  area of RPU, whereas  $T_{XRPU} = 2 \times T_{RPU}$  as  $\text{RPU}_{\pi_d}$  and  $\text{RPU}_{\pi_v}$  are in series. Using the estimates from Section 3.4 we estimated the area overhead and latency of XRPU for various commercial embedded processors and the results are tabulated in Table 3.3. In summary, the area overhead of this scheme is fairly insignificant. We see that for all the processors the area overhead is less than 1%. Even for low end embedded processor with  $10 \text{ mm}^2$  of area the overhead comes out to be 2.2% for 70 nm technology to 14.5% in 180 nm technology. The delay overhead is more easily hidden through pipelining. The overhead appears to be of the order of two cycles

Table 3.3 Latency and Area Overhead Estimation of XRPU

Processor	Technology	Package $mm^2$	Max Freq $MHz$	% Area Inc	Delay $ns$	Latency $cyc$
PXA 255 [30]	0.18 $\mu$ ,1.80V	17x17	400	0.50	3.367	2
PXA 26x [30]	0.18 $\mu$ ,1.30V	13x13	400	0.86	3.367	2
PXA 27x [30]	0.18 $\mu$ ,1.55V	13x13	624	0.86	3.147	2
PXA 800F [30]	0.13 $\mu$ ,1.20V	12x12	312	0.53		
PPC 750FX [29]	0.13 $\mu$ ,1.45V	21x21	900	0.17		
PPC 750CXe [29]	0.18 $\mu$ ,1.80V	27x27	700	0.20	3.367	3

for most of these technology nodes, and hence fits nicely into any pipeline.

### 3.6 Discussion

#### 3.6.1 Obfuscation Strength of Permutation Function (RPU)

In this section we quantify the *obfuscation strength* of the permutation function implemented using RPU. The aim of the permutation function is to obfuscate the instruction sequence. The first step in the process of reverse-engineering is disassembling the binary image. Once the instructions are disassembled their static sequence is used to reconstruct the control flow graph (CFG). Hence a measure of obfuscation could be derived from the dissimilarity between the original CFG and the CFG derived from the obfuscated static image.

The nodes in a CFG correspond to a basic block, a straight-line sequence of instructions. The permutation function rearranges the static instruction sequencing. This results in the modification of many basic blocks as constructed from the obfuscated/permuted image since there are no corresponding basic blocks in the original CFG. For instance, even if one instruction from an original CFG basic block is permuted away past a control instruction (a branch), a new basic block results in the obfuscated CFG. The edges in the permuted CFG similarly can either be completely new or may have a different source or target basic

block. We will call a basic block or edge perturbed if there is no corresponding basic block (in the way of graph isomorphism accounting for new naming) or edge in the original CFG.

A permutation function that perturbs all the nodes (basic blocks) and edges from the original CFG achieves *complete* obfuscation. We define an analytical limited version of this notion that captures the similarities of the instruction sequences in the original image versus the permuted image. We will estimate what fraction of sequences of  $n$  instructions are preserved (or perturbed) from original to the permuted image for a large range of values for  $n$ . A typical basic block is 5 to 10 instructions long. Such a measure for  $n = 5$  then estimates the fraction of perturbed basic blocks which constitutes a simplistic measure of obfuscation. We define such an obfuscation strength measure of size  $n$ ,  $OS_n$ , for the permutation function as follows.

**Definition 3.6.1.**

*Let*

$I \rightarrow$  *Unobfuscated binary image*

$I_{obf} \rightarrow$  *Obfuscated binary image*

$N \rightarrow$  *Number of instructions in  $I$  ( $|I| = |I_{obf}|$ )*

$S_n^j \rightarrow$  *Sequence of instructions  $i_1, i_2, \dots, i_n$  in  $I$   
from  $j^{th}$  position where  $1 \leq j \leq (N - n + 1)$*

*Then*

$OS_n =$  *% of  $S_n^j$  not in  $I_{obf}$*

Note that in our solution we permute the binary image in units of *words* (4 bytes). In some architectures (like x86) the instruction sizes are not fixed. Thus permutation could break some instructions giving rise to illegal or different instructions. Also the presence of static data in the image could cause the same effect. Hence this definition of *obfuscation strength* is very conservative and forms a lower bound.

To understand the definition of our *obfuscation strength* let us consider an example. Figure 3.5 shows an example permutation. In this example  $|I| = |I_{obf}| = N = 5$  and  $S_2^j$  exist for  $j = 1, 2, 3, 4$ . In  $I_{obf}$  only  $S_2^1$  exists unobfuscated. Hence  $OS_2 = 75\%$  and  $OS_n$  for  $n > 2$  is 100%.

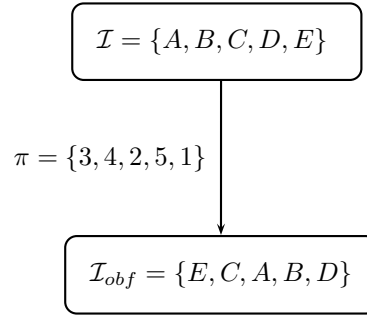


Figure 3.5 An Example Permutation

As explained in Section 2.4.3 RPU has 39 *configuration selection* bits. It is highly improbable to find the *obfuscation strength* of RPU by exercising all the  $2^{39}$  configurations. We generated  $2^{20}$  random configurations and found the average *obfuscation strength* for various sequence sizes. We repeated this experiment several times to get a statistical validation of our experiment. All the values are listed in Table 3.4

Table 3.4 Average *Obfuscation Strength* for  $2^{20}$  Runs

Seed	$OS_5$	$OS_6$	$OS_7$	$OS_8$	$OS_9$	$OS_{10}$	$OS_{11}$
0x031245f8	94.74	95.96	96.80	97.35	97.83	98.29	98.63
0x7fc5a2d5	94.75	95.98	96.82	97.37	97.84	98.30	98.64
0x015e8f8c	94.73	95.97	96.80	97.36	97.84	98.29	98.63
0x00231eea	94.74	95.97	96.80	97.36	97.83	98.29	98.63
0x0153d22e	94.75	95.97	96.81	97.36	97.84	98.30	98.64

We have listed the *obfuscation strength* for sequences of size from 5 to 11 as this happens to be the most frequent length for basic blocks. We see from Table 3.4 that with at least 95% probability our permutation function will obfuscate basic blocks with 5 or more instructions. This makes reverse-engineering of CFG from  $I_{obf}$  as difficult as the permutation function itself, which is  $\approx 2^{34}$  strong.

### 3.6.2 Attack Scenarios

A verification process using TIVA fails if one of following events occurs.

1. A malicious software executing on  $\mathcal{E}$  is able to generate the expected checksum  $\mathcal{H}$  in the expected response time  $\mathcal{T}$ .
2. An impostor system pretending to be  $\mathcal{E}$ , but with greater computational capabilities than  $\mathcal{E}$ , is able to generate the expected checksum  $\mathcal{H}$  in the expected response time  $\mathcal{T}$ .

For both these attacks to be successful the malicious software running on the device  $\mathcal{E}$  or the impostor system must know the composite permutation function  $\pi_d \cdot \pi_v$ . Since only microcode is able to exercise XRPU and it returns only the checksum value, it is not possible for the software running on the device to derive  $\pi_d$ . Thus the composite permutation function becomes as hard as the individual permutation functions which in our case is  $\approx 2^{34}$  strong.

If an impostor system gets hold of  $I_{obf}$  then it is possible to generate the required hash without the knowledge of  $\pi_d$ . Only  $\pi_v$  and  $I_{obf}$  suffice. This attack could be avoided in two ways. The verifier  $\mathcal{V}$  could make sure that  $I_{obf}$  is stored securely and trust the device vendor  $\mathcal{D}$  to not release  $I_{obf}$  to anyone else. Another method is to extend the verification protocol in the application layer to add a unique ID to the device  $\mathcal{E}$ .

Secure storage of secrets in device  $\mathcal{E}$  is essential for the functioning of TIVA. Any attack that could reveal  $\pi_d$  would de-obfuscate  $I_{obf}$  thus compromising its IP. However, storing secrets in hardware is a well researched topic and solutions like battery-backed RAM as used in IBM's 4758 [22] secure coprocessor could be used. Ishai et al. [31] proposed circuit techniques to protect circuits against probing attacks. This could be used to store the secret within the chip resistant to probing attacks.

### 3.6.3 Flexibility of TIVA

We explained TIVA for 10-bit permutation functions in a 32-bit architecture producing a 64-bit checksum, thus handling a memory size of 4KB. But TIVA is not restricted to this memory size. For bigger memory sizes the verification function could be easily extended to produce 64 bits for every chunk of 4KB. As we mentioned earlier the checksum or hash generation function proposed in our solution is not ideal.

It could be replaced with any other checksum generation function of any size. TIVA is not restricted to 32-bit architectures. It could very well be applied to 8-bit or 16-bit architectures. The microcode for the verification function could be modified accordingly.

TIVA is not restricted to Von Neumann or Harvard architectures either. As explained in [2] to verify Von Neumann architectures, in which program and data share the same memory, the device should be brought into some known state and then verification could be performed. This known state should be the one which is distributed as binary image to the verifier. Software vendors could distribute multiple such images for various checkpoints. In the case of Harvard architectures the program and data memory are separate hence it is sufficient to find the checksum of program memory alone.

### 3.7 Related Work

Seshadri et al. proposed software only attestation mechanism in SWATT ([2]). A software only solution will incur lower cost than an attestation technique that requires additional hardware. It can also be used on legacy systems. These were the two major selling points for SWATT. But SWATT is probabilistic, i.e. it accesses the memory based on a pseudo-random sequence. The verification procedure performs  $O(n \log n)$  memory accesses, for memory size  $n$ , in order to access all of the memory with high probability. They generate 16-bit addresses from an 8-bit RC4 pseudo-random number by adding to it the current value of checksum. This could very well affect the probability distribution of the PRG sequence. The effect of this change on the probability of accessing every memory location in the system is not studied in the paper. Additionally, embedded devices in most cases are limited by battery power. Deployment of such a probabilistic method will incur energy penalty.

Kennell et al. proposed software only solution *Genuinity* ([45]) to address the problem of autonomous integrity verification of remote systems. This solution is applicable only to general purpose systems which expose architectural parameters like TLB miss counters, etc. They used these architectural side effects to uniquely generate a checksum through the verification procedure. They argued that this checksum cannot be generated whenever the verification procedure is modified or through other emulated/simulated systems. But Shankar et al.([47]) proved that such a system based on architectural side effects is not sufficient to authenticate software.

Other solutions like IBM's IMA [46] use trusted hardware support [52] and require sophisticated OS support to verify the integrity. TPM provides root of trust for storage, for measurement, and for reporting. But TPM requires Public Key Infrastructure (PKI) and support of sophisticated message authentication algorithms like HMAC ([27]) to provide these trusts. The requirements of TPM and sophisticated OS support may be more than what an embedded device could offer. Moreover in IBM's IMA integrity verification is done only at the loading point. Hence any attack that occurs after the software is loaded will not be captured. Also the verifier is assumed to know the hash of the software or system configuration being verified. This again breaks our requirement of IP protection.

Thus earlier proposed solutions did not recognize IP protection as an important dimension in the problem of integrity verification. Our solution, TIVA, is different from these solutions in various aspects, such as

- TIVA uses a hardware component to aid the verification.
- TIVA is deterministic, i.e. it accesses each memory location at most once during verification.
- TIVA uses a shared secret between the embedded device and verifier to make simulating/emulating the device very difficult.
- TIVA uses permutation function to achieve both IP protection and randomness in hash generation function.

### **3.8 Conclusions**

Embedded devices are omnipresent and pervade all facets of human life. This penetration is likely to only increase in the future. Their sheer numbers and wide presence make them amenable to tampering. A tampered sensor could misrepresent its environment (report no bio-hazard particles where there are some) or a tampered PDA could relay the private data of the user to a third party. Hence verification of these devices is a relevant problem. However, such verification needs to be extremely efficient and mostly automated given the sheer numbers of these devices. Moreover, the verification architecture will not be practical if it compromises the IP of the software running on these devices. We present a novel hardware architecture TIVA and a schema for such a verification mechanism which satisfies all the requirements

of a verification system without compromising the IP of the system being verified. We demonstrate that the silicon area overhead for TIVA is minimal, 1%, and its time overhead is completely absorbed in the pipeline.

## CHAPTER 4. REBEL: RECONFIGURABLE BLOCK ENCRYPTION LOGIC

Existing block cipher function designs have tended to deploy the secret bits in a specific and limited way. We generalize the role of the secret as truth tables of Boolean gates in a carefully designed logic schema. Our claims are: these reconfigurable functions are pseudo one-way, and pseudo random functions. Such a function family is proposed using reconfigurable gates. Based on this function family we create *REBEL*, REconfigurable Block Encryption Logic, which is an LR-Network, and prove its cryptographic and cryptanalytic security. From cryptographic perspective, this function appears to be a pseudo-permutation. From cryptanalysis perspective, any observable attribute appears to be a random process.

### 4.1 Introduction

Conventional block ciphers ([41, 40]) derive their security from an embedded secret, more commonly referred to as a *key*. One of the inputs, key, in each round is secret whereas the round functions themselves are public. This is a deliberate design decision so that the algorithm can be published as a standard. The secret, however, is combined with the state in a limited way, as an xor, during a round. The xor based mixing of the cipher state and the secret leads to some vulnerabilities based on linear and differential cryptanalysis. The complexity of extracting the secret or its properties is proportional to the non-linearity, among many other attributes, of the round functions.

We propose a simple yet novel approach wherein the round functions themselves become the secret, while the function schema is a publicly published algorithm. The intuition is to use reconfigurable gates as round functions and define their configurations as the secret (or key). Hence the complexity of such a cryptographic function is derived from the fact that almost all of the round processing is driven by the secret (truth tables). In a traditional block cipher, the secret is combined with the state with an xor as one

of the steps in the round. This xor step is susceptible to linear modeling of the secret and input/output relationship. When the secret is used as a Boolean gate truth table, it is inherently non-linear, especially when many levels of such gates are composed, and when each gate is selected to have large minterm degree.

The advantage of such a reconfigurable block encryption logic (*REBEL*) is that it is highly *time-efficient* when implemented in hardware. The throughput needs of cryptographic blocks when placed in-line into a processor pipeline in order to support secure execution environments [34, 50]. Another added advantage is that the the key-space is much larger than the block length of the encryption function. This provides much higher security levels without having to increase the block length.

## 4.2 Preliminaries

### 4.2.1 Notations

- $I_n$  denotes the set of all  $n$ -bit strings,  $\{0, 1\}^n$ .
- For  $x \in I_{2n}$ , denote by  $x|_L$  the first (left)  $n$  bits of  $x$  and by  $x|_R$  the last (right)  $n$  bits of  $x$ .
- $G_n^b$  denotes the set of all  $I_n \mapsto I_1$  balanced gates (functions).
- Let  $x$  and  $y$  be two bit strings of equal length, then  $x \oplus y$  denotes their bit-by-bit exclusive-or.
- Let "o" denote function composition operator, *i.e.*,  $f = g \circ g$  implies,  $f(x) = g(g(x))$ .
- Let "•" denote concatenation of binary digits, *i.e.*,  $p = x \bullet y$  implies,  $p$  is  $2N$  bit string with left  $N$  bits containing  $x$  and right  $N$  bits containing  $y$ .
- Let  $f^{(n)}$  represent the composition of function  $f$   $n$ -times *i.e.*, for example  $f^{(3)} = f \circ f \circ f$ .
- Let a gate  $g$  be defined over  $n$  input bits/variables  $\{x_1, x_2, \dots, x_n\}$ . A literal  $l$  is defined as either an input variable  $x_i$  or its complement  $\bar{x}_i$ . A minterm is defined as a product of  $n$  literals wherein no variable  $x_i$  occurs both in complemented and uncomplemented literal form.
- $\in_R$  represents the action of choosing uniformly at random an element from a set.

## 4.2.2 Properties of $G_n^b$

### 4.2.2.1 Cardinality:

$$|G_n^b| = \binom{2^n}{2^{n-1}}.$$

### 4.2.2.2 Closed over complement:

**Definition 4.2.1.** A set of gates  $G$  is closed over complement if  $\forall g \in G$  the complement  $\bar{g}$  is also present in  $G$ .

Consider a gate  $g \in G_n^b$ . Then  $g$  is balanced and has equal number of zeros and ones in its truth table. The truth table of its complement gate  $\bar{g}$  is nothing but the complement of every truth table row of  $g$ , i.e., every zero of  $g$  will become one and every one of  $g$  become zero. Hence the number of zeros and ones in  $\bar{g}$  is still the same as  $g$ . Hence  $\bar{g}$  is also balanced and is present in the set  $G_n^b$ . Thus the set  $G_n^b$  is closed over complement.

### 4.2.2.3 Symmetry over Input Variables:

Any gate  $g$  can be represented as a boolean function of input variables. Let the set of  $n$  input variables to the gate  $g$  is  $I(g) = \{x_1, x_2, \dots, x_n\}$ . The support set of  $g$ ,  $Sup(g) \subseteq I(g)$ , which is the set of all input variables  $g$ 's value depends on, i.e., if  $x_i \in Sup(g)$  then  $\delta g / \delta x_i \neq 0$ . A set of gates  $G$  is symmetric if for each  $g \in G$ , if any pair of input variables in  $Sup(g)$  are swapped in the expression for  $g$  leading to a gate  $g'$ , then  $g' \in G$ . This property exhibits  $G$ 's bias towards certain input variables. For example if the set  $G$  has more gates with variable  $x_i$  than any other variables then the set becomes biased towards  $x_i$  and any change in  $x_i$  is reflected at the output with higher probability than a change in any other input variables.

**Definition 4.2.2.** A set of gates  $G$  is said to be symmetric in input variables if  $\forall g \in G$  the gate  $g'$  obtained by swapping any two input variables from the support set of  $g$  is also present in  $G$ .

For  $g \in G_n^b$ , the balance property dictates that there be  $2^{n-1}$  minterms. A swap of  $x_i$  and  $x_j$  in the expression for  $g$  does not alter the number of 1's (minterms). It merely, recodes the minterm locations. Hence the gate  $g'$  derived from  $g$  through a swap of two variables will also have  $2^{n-1}$  minterms, and hence

will be balanced, implying  $g' \in G_n^b$ . Hence, the set  $G_n^b$  is *symmetric over input variables*. Now on, we will use the term symmetric set  $G$  to mean a set symmetric in input variables.

#### 4.2.2.4 Closed over variable-complement:

**Definition 4.2.3.** We define a set of gates  $G$  as closed over input variable-complement if  $\forall g \in G$  the gate  $g'$  obtained by complementing all literal instances of an input variable  $x_i$  is also present in  $G$ .

Let  $g \in G_n^b$  and let  $g'$  be obtained by complementing the variable  $x_i$ . Since  $g$  is balanced the number of 1's (minterms) is  $2^{n-1}$ . Each affected minterm gets relocated to a different row in the truth table through complementing of a literal  $x_i$  or  $\bar{x}_i$ . However, the number of minterms still remains at  $2^{n-1}$ , and hence  $g'$  is also balanced. Thus  $G_n^b$  is *closed over input variable-complement*.

#### 4.2.2.5 input-collision probability:

**Definition 4.2.4.** For any gate  $g$ , the input-collision probability  $p_{coll}^g$  is defined as the probability that any pair of inputs  $x, x' \in_R I_n$  s.t.  $x \neq x'$ , produce the same output, i.e.,  $p_{coll}^g = P[g(x) = g(x') | g]$ . Note that the collision probability is averaged over all the input pairs.

For every  $g \in G_n^b$  there are equal number ( $2^{n-1}$ ) of 0's and 1's. For the collision to happen, the input  $x$  has  $2^n$  out of  $2^n$  choices and the input  $x'$  has  $2^{n-1} - 1$  out of  $2^n - 1$  choices. Hence  $p_{coll}^g = \frac{2^{n-1}-1}{2^n-1}$ .

#### 4.2.2.6 gate-collision probability:

**Definition 4.2.5.** For any given input  $x \in I_n$  gate-collision probability is defined as the probability that any pair of gates  $g, g' \in_R G$  collide i.e.,  $p_{gcoll}^G(x) = P[g(x) = g'(x) | x]$ . Note that the collision probability is averaged over all the gate pairs.

The set  $G_n^b$  is *closed over complement*. Hence for every  $g \in G_n^b$  the complement gate  $\bar{g}$  is also present in  $G_n^b$ . Thus every row of the truth table for the set is also balanced and  $p_{gcoll}^G(x) = \frac{1}{2}$ .

#### 4.2.2.7 set-collision probability:

**Definition 4.2.6.** For any set of gates  $G$ , the set-collision probability  $p_{coll}^G(x, x')$  is defined as the probability that any gate  $g \in_R G$  collides for any given pair of inputs  $x, x' \in I_n$  s.t.  $x \neq x'$ , i.e.,  $p_{coll}^G(x, x') = P[G(x) =$

$G(x') | x, x']$ , where  $G(x) = \frac{\sum_{g \in G} g(x)}{|G|}$ . Note that the collision probability is averaged over all the gates in the set  $G$ .

In  $G_n^b$  for every input pair  $x, x'$  the number of gates that collide are  $N_{coll} = 2 \times \binom{2^n - 2}{2^{n-1} - 2}$  and the number of gates that does not collide are  $N_{\overline{coll}} = 2 \times \binom{2^n - 2}{2^{n-1} - 1}$ . Hence the probability of collision is

$$p_{coll}^G = \frac{N_{coll}}{N_{coll} + N_{\overline{coll}}} = \frac{2^{n-1} - 1}{2^n - 1}.$$

In other words, the set  $G_n^b$  is symmetric with respect to every pair  $x, x'$ .

#### 4.2.2.8 input symmetric-controllability:

**Definition 4.2.7** (input pairs equivalence class). Let  $p, p' \in I_n$  be the inputs to the gates in the set  $G$ . We group the pairs of inputs based on their hamming distance, i.e., we define equivalence class as  $S_d = \{(p, p') | h(p, p') = d\}$ .

**Definition 4.2.8.** A set of gates  $G$  is input symmetric-controllable if the collision probability is same for any pair of inputs in the same equivalence class.

**Lemma 4.2.1.** Any symmetric set of gates  $G$  and closed over variable complement is input symmetric-controllable.

*Proof.* Let  $(p, p'), (q, q') \in S_d$  then the input-symmetry property requires that  $P[G(p) = G(p')] = P[G(q) = G(q')]$ . The trivial cases are  $p = q, p' = q'$  and  $p = q', p' = q$ . We will prove the non-trivial cases. Let  $i_1, i_2, \dots, i_d$  s.t.  $i_1 < i_2 < \dots < i_d$  represent the  $d$  bit-positions in which the pair  $(p, p')$  differ and similarly  $j_1, j_2, \dots, j_d$  s.t.  $j_1 < j_2 < \dots < j_d$  represent the  $d$  bit-positions in which the pair  $(q, q')$  differ.

**Case**  $\{i_1, i_2, i_3, \dots, i_d\} = \{j_1, j_2, j_3, \dots, j_d\}$ :

Let  $k_1, k_2, \dots, k_{n-d}$  be the bit-positions which do not differ in both the pairs. Then  $p$  and  $q$  should differ in at least one of these bit-positions. Let  $g \in G$  be the gate that collides for the pair  $(p, p')$ . Let  $g'$  be a gate obtained by changing the literals of  $g$  in the following way.  $x_{k_l}^{g'} = x_{k_l}^g \oplus p_{k_l} \oplus q_{k_l}$ , that is, the literals are complemented in bit-positions where  $p$  and  $q$  differ. But  $G$  is closed over literal-complement. Hence  $g'$  is also present in the set  $G$  and  $g'$  will collide for the pair  $(q, q')$ .

**Case**  $\{i_1, i_2, i_3, \dots, i_d\} \neq \{j_1, j_2, j_3, \dots, j_d\}$ :

Let  $g \in G$  be a gate that collides for the pair  $(p, p')$ . Let  $g'$  be a gate such that the input variables are swapped in the following way. Variables  $x_{i_l}^{g'} = x_{j_l}^g$ ,  $x_{j_l}^{g'} = x_{i_l}^g$  that is the variables at positions  $i_k$  and  $j_k$  are swapped. But the set  $G$  is *symmetric*. Hence  $g'$  is also present in the set  $G$  and  $g'$  will collide for the pair  $(y, y')$ .

Thus for every gate  $g$  that collides for the pair  $(p, p')$ , a gate  $g'$  that collides for the pair  $(q, q')$ , exists. Hence the probability of collisions are same or  $P[G(p) = G(p')] = P[G(q) = G(q')]$ . □

**Corollary 4.2.1.**  $G_n^b$  is input symmetric-controllable.

*Proof.*  $G_n^b$  is symmetric and closed over variable-complement (Properties 4.2.2.3 and 4.2.2.4 ). □

#### 4.2.2.9 *k*-wise set-collision probability:

Let  $g_i$  represent  $i^{th}$  gate that is chosen independently at random from  $G_n^b$ . Let  ${}^k p_{coll}^G = \prod_{i=1}^k P[g_i(x) = g_i(x')]$  be the probability that all the  $k$  gates collide for any given pair of inputs  $x, x' \in I_n$  s.t.  $x \neq x'$ . Then  ${}^k p_{coll}^G = (p_{coll}^G)^k$ .

From Property 4.2.2.7 for every pair of inputs  $x$  and  $x'$  number of gates that collide in the set  $G_n^b$  is  $N_{coll}$ . That is, the probability of picking a gate from  $G_n^b$  such that it collides for any input pair  $x$  and  $x'$  is  $\frac{N_{coll}}{|G_n^b|}$ . And this probability is same for all the choices of the gates since they are chosen independently and at random. Hence  ${}^k p_{coll}^G = (p_{coll}^G)^k$ .

#### 4.2.2.10 bias propagation:

Let  $x \in I_n$  be the input to gate  $g \in G_n^b$ . Let  $x_i$  be the  $i^{th}$  bit. Let  $\epsilon_{in}$  be the bias of the input bits *i.e.*, the probability that the input bit  $i$  collides is  $\frac{1}{2} + \epsilon_{in}$  and  $-\frac{1}{2} \leq \epsilon_{in} \leq \frac{1}{2}$ . Then the bias at the output  $\epsilon_{out}$  is

$$\epsilon_{out} = \left(\frac{1}{2} + \epsilon_{in}\right)^n + \left(1 - \left(\frac{1}{2} + \epsilon_{in}\right)^n\right) \cdot \left(\frac{2^{n-1} - 1}{2^n - 1}\right) - \frac{1}{2} \quad (4.1)$$

In order to find how the *output bias* behaves with respect to the *input bias* or the *bias propagation* we

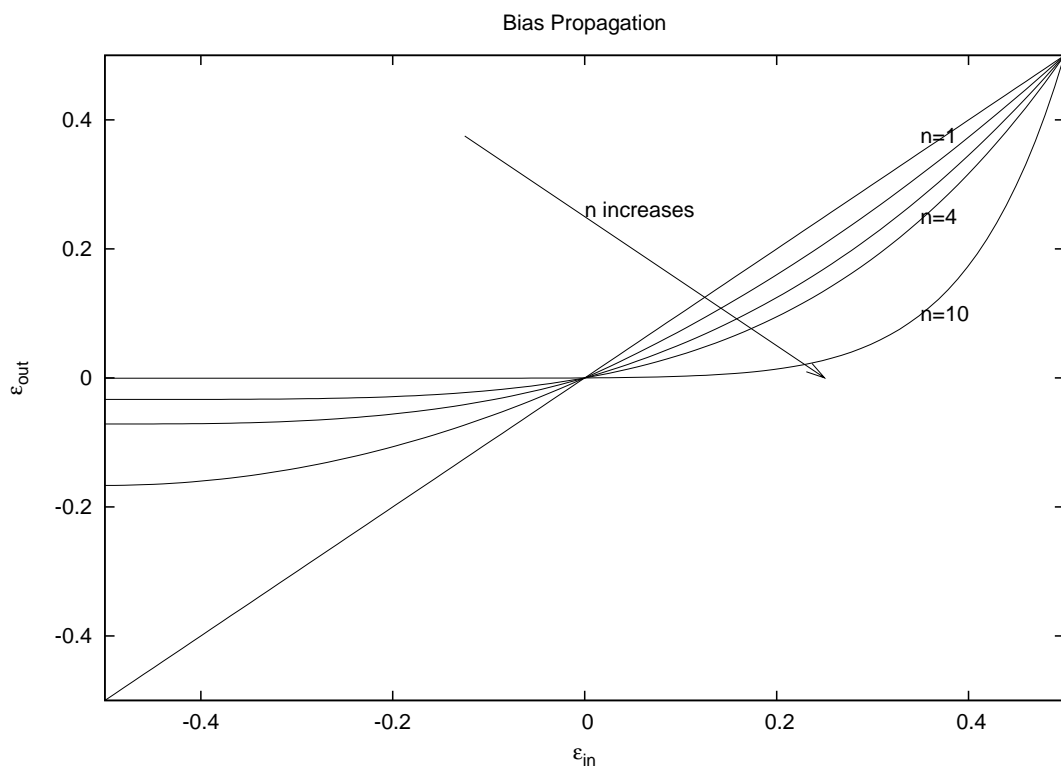


Figure 4.1 Bias Propagation in  $n$ -to-1 Gate

differentiate  $\epsilon_{out}$  with respect to  $\epsilon_{in}$ . We get,

$$\begin{aligned} \frac{\partial \epsilon_{out}}{\partial \epsilon_{in}} &= n \cdot \left( \epsilon_{in} + \frac{1}{2} \right)^{n-1} \cdot \left( \frac{2^{n-1}}{2^n - 1} \right) \\ &\geq 0 \end{aligned}$$

Thus the slope is a *strictly increasing* function in  $\epsilon_{in}$ , and,  $\frac{\partial \epsilon_{out}}{\partial \epsilon_{in}} \Big|_{\epsilon_{in}=0} \leq 1$ . Thus,

$$|\epsilon_{out}| \leq |\epsilon_{in}| \quad (4.2)$$

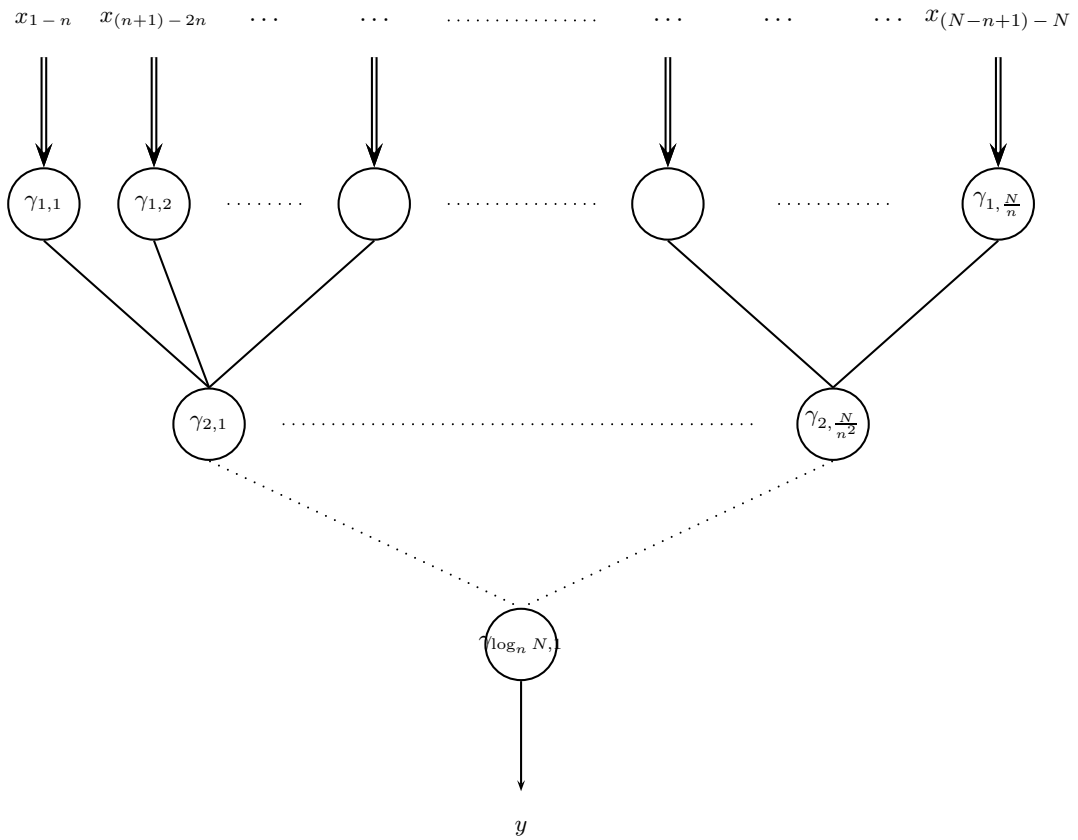
In other words the *bias* decreases as it propagates through the gate. Figure 4.1 illustrates this phenomenon. The figure has *output bias* plotted against *input bias* for many values of  $n$ . The interesting thing to observe is, as  $n$  increases the *bias* propagation is very sharp or, the *output bias* stays closer to 0 for wider range of *input bias*.

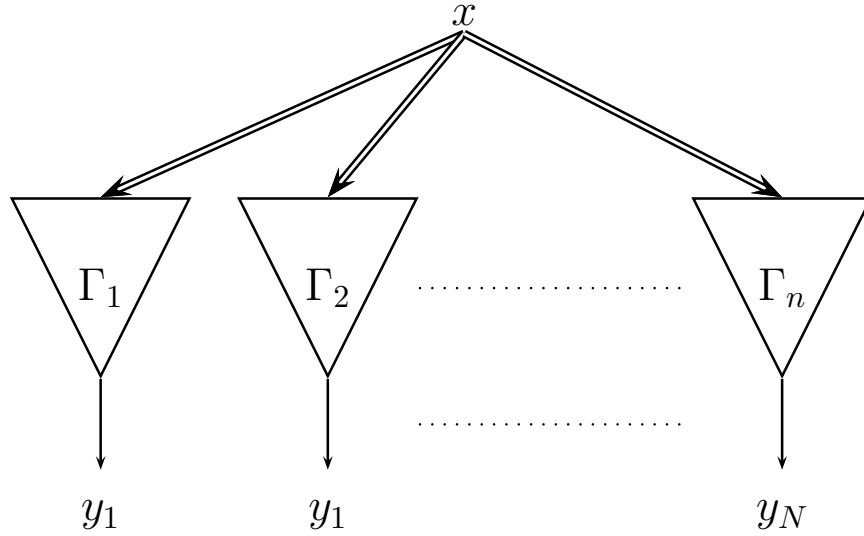
### 4.3 Function Family $F_n^N$

**Definition 4.3.1** ( $F_n^N$  function family).  $F_n^N$  is a  $I_N \mapsto I_N$  family of functions that uses  $I_n \mapsto I_1$  reconfigurable gates as the component gates, where,  $N = n^{\lfloor \log_n N \rfloor}$ .

#### 4.3.1 Notations

- Let  $\gamma$  be a reconfigurable gate.
- Let  $x(\gamma)$  and  $y(\gamma)$  be the input and output of the reconfigurable gate respectively.
- Let  $g(\gamma)$  be the gate implemented by (or the truth table of) the reconfigurable gate.
- Let  $\Gamma$  be a  $I_N \mapsto I_1$  tree using  $I_n \mapsto I_1$  reconfigurable gates (or  $n$ -ary tree). Then there are  $\log_n N$  levels in  $\Gamma$  (wlog let top level be 1) and  $i^{th}$  level has  $\frac{N}{n^i}$  reconfigurable gates (wlog let leftmost gate be 1). Figure 4.2 shows the diagrammatic representation of this construction.
- Let  $N_i^\Gamma$  be the number of gates present at  $i^{th}$  level.
- Let  $\gamma_{i,j}(\Gamma)$  be the reconfigurable gate at  $i^{th}$  level  $j^{th}$  position.
- Let  $\uparrow$  represent switching of any bit  $b$  i.e.,  $P[b \uparrow]$  represent the probability that the bit  $b$  switches.

Figure 4.2 Diagrammatic representation of  $\Gamma$

Figure 4.3 Diagrammatic representation of  $F_n^N$ 

### 4.3.2 Construction of $F_n^N$

$F_n^N$  is a  $I_N \mapsto I_N$  family of functions that uses  $I_n \mapsto I_1$  reconfigurable gates as the component gates.  $F_n^N$  has  $N$  tree-gates ( $\Gamma$ ) each one producing one output bit using the same input bits. Let  $\Gamma_i$  be the  $i^{\text{th}}$  tree-gate (*wlog* let leftmost be 1). Figure 4.3 shows this construction. In the figure,  $x$  represents the  $N$  bit input and  $y_i$  represent the  $i^{\text{th}}$  bit of  $y$  which is the  $N$  bit output.

The configurations of the reconfigurable gates are connected as  $g(\gamma_{i,j}(\Gamma_k)) = g(\gamma_{p,q}(\Gamma_r))$  where,

$$\left(\sum_{t=1}^{i-1} N_t^\Gamma + j + k - 2\right)\%N = \left(\sum_{t=1}^{p-1} N_t^\Gamma + q + r - 2\right)\%N$$

and  $1 \leq i, p \leq \log_n N$ ,  $1 \leq j \leq \frac{N}{n^i}$ ,  $1 \leq q \leq \frac{N}{n^p}$ , and  $1 \leq k, r \leq N$ . Thus only  $N$  unique gate configurations are required by  $F_n^N$ . The configurations are chosen uniformly at random from the set  $G_n^b$ . The configurations forms the key to function family.

Thus the key space is  $\mathcal{K} = \{g \mid g \in G_n^b\}^N$  and its cardinality is  $|\mathcal{K}| = (|G_n^b|)^N$ . Let  $\kappa \in \mathcal{K}$  and  $\kappa_i$  be the  $i^{\text{th}}$  gate configuration. Then the gate configuration assignments are done as,  $g(\gamma_{p,q}(\Gamma_r)) = \kappa_i$  where  $i = 1 + \left(\sum_{t=1}^{p-1} N_t^\Gamma + q + r - 2\right)\%N$  and  $1 \leq p \leq \log_n N$ ,  $1 \leq q \leq \frac{N}{n^p}$ , and  $1 \leq i, r \leq N$ .

The construction can be explained in simpler terms as follows. Consider  $N$  columns and  $\frac{N-1}{n-1}$  rows of reconfigurable gates. Then  $i^{\text{th}}$  column represents the gates of the tree-gate  $\Gamma_i$  and  $j^{\text{th}}$  row in a column

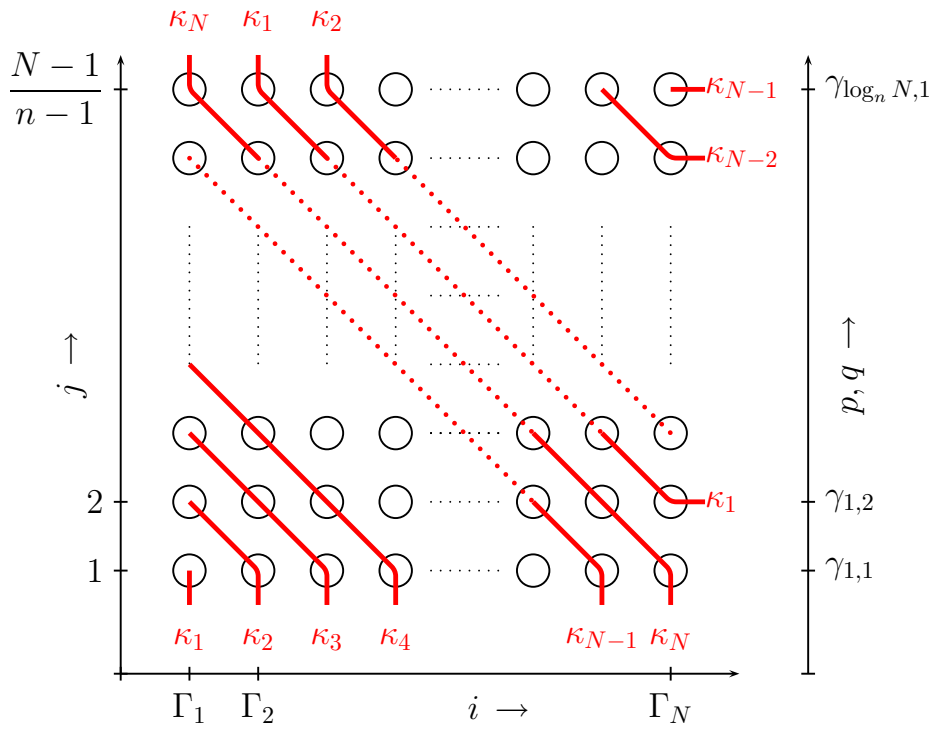


Figure 4.4 Construction of  $F_n^N$  and Key (Gate Configuration) Assignment

corresponds to the  $q^{th}$  gate in  $p^{th}$  level such that  $j = \sum_{t=1}^{p-1} N_t^\Gamma + q$ . In every row there are  $N$  reconfigurable gates and are assigned  $N$  gates from the key i.e., if  $\gamma_{i,j}$  represent the reconfigurable gate in  $i^{th}$  column and  $j^{th}$  row, then  $g(\gamma_{i,j}) = \kappa_{(i+j-2)\%N+1}$ . Figure 4.4 shows this construction diagrammatically.

### 4.3.3 Properties of $F_n^N$

**Definition 4.3.2 (input group).**  $N$  bits of input  $x$  to  $F_n^N$  are divided into equal sized groups of size  $n$  bits. Bits of group  $-i$  of  $x$ , denoted by  $grp(x, i)$ , correspond to the input bits of  $i^{th}$  reconfigurable gate in level-1 (top level), bits  $x_{(i-1)*n+1}, x_{(i-1)*n+2}, \dots, x_{i*n}$ .

**Definition 4.3.3 ( $\Delta_g$ -difference operator).**  $\Delta_g$  is a difference operator that takes two  $N$  bit values and returns the number of groups in which the two inputs differ; i.e.,  $\Delta_g(x, x') = i$ , where  $i \in [0, \frac{N}{n}]$  and  $x, x' \in I_N$ . Let  $\delta_g(x, x') = \{i_1, i_2, \dots, i_k\}$  such that  $0 \leq k \leq \Delta_g(x, x')$  be the set of group indices where  $x$  and  $x'$  differ.

**Definition 4.3.4 (group hamming distance operator).** Given two  $N$  bit values  $x, x' \in I_N$  the group hamming distance operator  $h_{\Delta_g}$  is defined as,  $h_{\Delta_g}(x, x') = [h_{i_1}, h_{i_2}, \dots, h_{i_{N/n}}]$ , the sequence of groupwise Hamming distances in ascending order, i.e.,  $h_{i_m} = h(grp(x, i_m), grp(x', i_m))$  and  $h_{i_p} \leq h_{i_q}$  for  $p < q$ .

#### 4.3.3.1 bit-Balance:

Every output bit of every function  $f \in F_n^N$  is balanced. This property is derived from the fact that every output is produced by a  $n$ -ary tree of balanced gates. Let  $x, y \in I_N, f \in F_n^N, y = f(x)$  and  $y_i$  be the  $i^{th}$  output bit. When averaged over all the inputs to the tree-gate  $\Gamma_i$ ,  $P[\Gamma_i(x) = 0] = P[\Gamma_i(x) = 1]$ , hence  $P[y_i = 1] = P[y_i = 0]$ .

#### 4.3.3.2 key-collision probability:

**Definition 4.3.5.** For the function family  $F_n^N$  key-collision probability  $p_{kcoll}$  is defined as the probability that for a given input  $x \in I_N$  any pair of keys  $\kappa, \kappa' \in_R \mathcal{K}$  produce the same output i.e.,  $p_{kcoll}(x) = P[f_\kappa(x) = f_{\kappa'}(x) | x]$ . Note that the probability is averaged over all the pairs of keys  $\kappa, \kappa'$ .

To derive this property let us consider each tree individually. Let  $\Gamma_i^\kappa$  and  $\Gamma_i^{\kappa'}$  be the tree  $i$  with keys  $\kappa$  and  $\kappa'$  respectively. Then the output of trees collide if  $\gamma_{\log_n N, 1}(\Gamma_i^\kappa) = \gamma_{\log_n N, 1}(\Gamma_i^{\kappa'})$ . Even though inputs

to the trees are the same the inputs to the root node are independent. This follows from the *gate-collision* property (Property 4.2.2.6) of  $G_n^b$ . Consider two level 1 gates  $\gamma_{j,1}(\Gamma_i^K)$  and  $\gamma_{j,1}(\Gamma_i^{K'})$ . Let  $b_j$  and  $b'_j$  be their outputs respectively. Then,  $P[b_j = x \cap b'_j = y] = P[b_j = x] \cdot P[b_j = y]$  where  $x, y \in \{0, 1\}$ . This is because,

$$P[b_j = 0 \cap b'_j = 0] = P[b_j \text{ and } b'_j \text{ collide}] \cdot P[b_j = 0] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

$$P[b_j = 0 \cap b'_j = 1] = P[b_j \text{ and } b'_j \text{ not collide}] \cdot P[b_j = 0] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

$$P[b_j = 1 \cap b'_j = 0] = P[b_j \text{ and } b'_j \text{ not collide}] \cdot P[b_j = 1] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

$$P[b_j = 1 \cap b'_j = 1] = P[b_j \text{ and } b'_j \text{ collide}] \cdot P[b_j = 1] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

Thus output bits of every level 1 gates  $\gamma_{j,1}(\Gamma_i^K)$  and  $\gamma_{j,1}(\Gamma_i^{K'})$  are independent. Hence the input bits to the root node are independent. Let  $g_\gamma = g(\gamma_{\log_n N, 1}(\Gamma_i^K))$  and  $g'_\gamma = g(\gamma_{\log_n N, 1}(\Gamma_i^{K'}))$  and let  $x_\gamma$  and  $x'_\gamma$  be the inputs to these gates respectively. Then, probability of *key-collision* for a tree is given by  $P[g_\gamma(x_\gamma) = g'_\gamma(x'_\gamma)]$ . Since both the gates  $g_\gamma$  and  $g'_\gamma$  are balanced the probability of collision is nothing but  $\frac{1}{2}$ . Since root nodes of every tree are independent, for any input  $x$ ,  $p_{kcoll}(x) = \frac{1}{2^N}$ .

Thus for any input  $x \in I_N$

$$p_{kcoll}(x) = \frac{1}{2^N} \tag{4.3}$$

Another property which follows the *key-collision* property is the probability that  $k$  inputs  $x_1, x_2, \dots, x_k$  chosen uniformly at random from  $I_N$  collide *i.e.*,  $f_K(x_i) = f_{K'}(x_i) \forall 1 \leq i \leq k$ . Since every input is independent this is nothing but  $\frac{1}{2^{Nk}}$ .

### 4.3.3.3 Tree independence:

For any function  $f \in_R F_n^N$  the collision probability for any given pair of inputs,  $x, x' \in I_N$  *s.t.*,  $x \neq x'$  is  $P[f(x) = f(x') | (x, x')] = \prod_{i=1}^k P[\Gamma_i(x) = \Gamma_i(x') | (x, x')]$ . *Wlog* let  $x$  and  $x'$  differ in  $k$  groups denoted by  $i_1, i_2, \dots, i_k$ . From the *k-wise set-collision* property of  $G_n^b$  (Property 4.2.2.9) the collision probabilities of  $N$  level-1 gates in each of the groups  $i_1, i_2, \dots, i_k$  are independent. Similarly the collision probabilities of

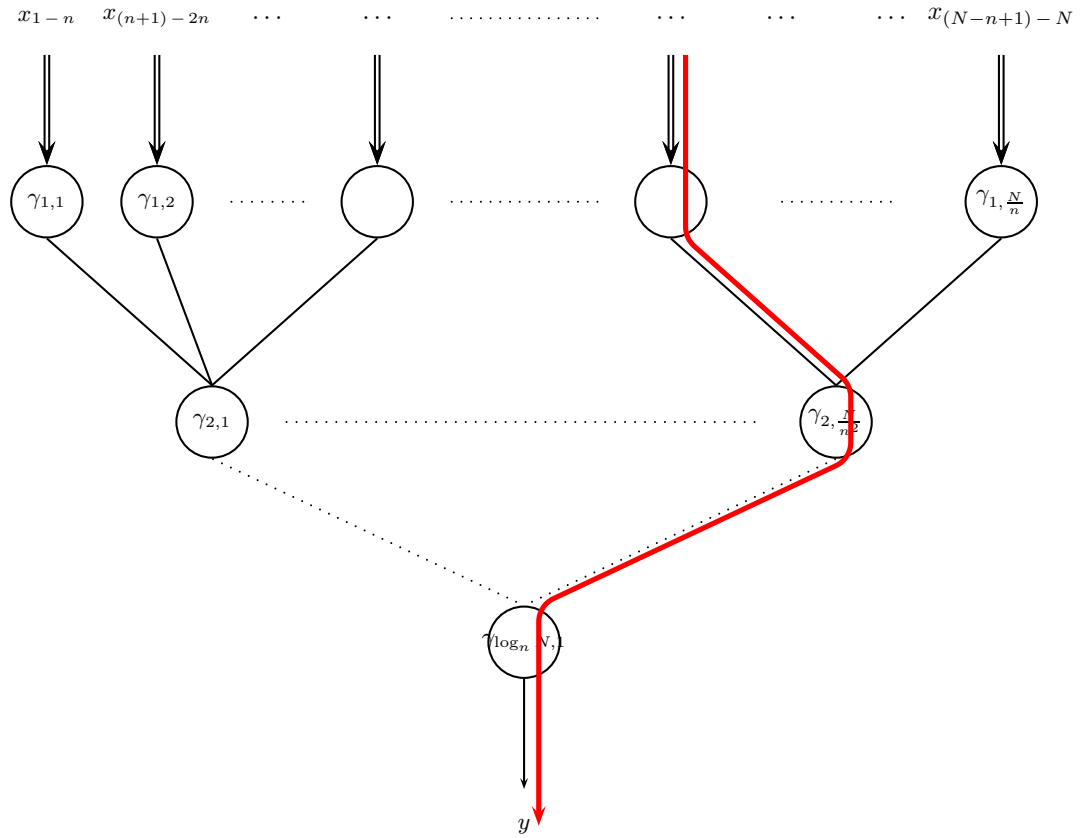


Figure 4.5 One of the Maximum Controllability Paths in  $\Gamma$

$N$  gates in the same group in subsequent levels are also independent. Hence the collision probabilities of every tree becomes independent. Thus,

$$P[f(x) = f(x') | (x, x')] = \prod_{i=1}^N P[\Gamma_i(x) = \Gamma_i(x') | (x, x')]$$

Since each tree is similar in structure, the collision probabilities of every tree is same. Thus,

$$P[f(x) = f(x') | (x, x')] = (P[\Gamma(x) = \Gamma(x') | (x, x')])^N$$

#### 4.3.3.4 Collision controllability:

**Definition 4.3.6.** Collision controllability  $\delta_c$  is the maximum probability with which any two inputs  $x, x' \in I_N$  to the function  $f \in_R F_n^N$  cause collision, i.e.,  $\delta_c = \max_{x, x'} \{P[f(x) = f(x')]\}$ .

From Property 4.3.3.3 the collision probabilities of trees are independent and equal. Hence to maximize the collision probability of  $f$ , the collision probability of tree has to be maximized. The collision probability of the tree gate will be maximum if inputs to only one gate on the top level switch, i.e., requiring minimum number of gates in the tree to collide. Figure 4.5 shows one such path in  $\Gamma$ . Thus,

$$\begin{aligned} \max_{x, x'} \{P[\Gamma_i(x) = \Gamma_i(x')]\} &= 1 - (1 - p_{coll}^g)^{\log_n N} = 1 - \left(\frac{2^{n-1}}{2^n - 1}\right)^{\log_n N} \\ \implies \delta_c &= \left(1 - \left(\frac{2^{n-1}}{2^n - 1}\right)^{\log_n N}\right)^N \end{aligned} \quad (4.4)$$

#### 4.3.3.5 function family collision probability:

**Definition 4.3.7.** Function family collision probability is the probability that any two inputs  $x, x' \in_R I_N$  s.t.  $x \neq x'$  cause collision in any function  $f \in_R F_n^N$  i.e.,  $p_{coll}^f = P[f(x) = f(x')]$ . Note that the probability is averaged over all the input pairs and all the functions in the family.

Let  $i = \Delta_g(x, x')$  and  $p_{group}^i$  be the probability that the inputs with  $\Delta_g = i$  cause collision. Let  $N_{group}^i$  be number of inputs with  $\Delta_g = i$ . Then,

$$p_{coll}^f = \frac{\sum_{i=1}^N N_{group}^i p_{group}^i}{\frac{2^N (2^N - 1)}{2}} \quad (4.5)$$

#### 4.3.3.6 input-symmetry:

The function family  $F_n^N$  exhibits the following input symmetry. Let  $x, x', y, y' \in I_N$ ,  $f \in F_n^N$  and  $y = f(x), y' = f(x')$ . Let  $y_i, y'_i$  be the  $i^{th}$  output bit of  $y$  and  $y'$  respectively. Then,

$$P[y_i = y'_i | (x, x')] = P[y_j = y'_j | (x, x')]$$

That is, the probability of collision for every output bit is same for any pair of inputs when averaged over all the functions in the family  $F_n^N$ .

**Definition 4.3.8.** We define the property input-symmetry as the probability of collision of every output bit being equal for any input change.

The trivial case is when  $x = x'$ . We will see the non-trivial case when  $x \neq x'$ . The collision probability of an output bit is nothing but the collision probability of the corresponding tree  $\Gamma_i$ . From Property 4.3.3.3 the collision probabilities of trees are independent and equal for a given pair of inputs. Hence,  $P[y_i = y'_i | (x, x')] = P[y_j = y'_j | (x, x')]$  for any pair of inputs  $x, x'$ .

#### 4.3.3.7 output-symmetry:

The function family  $F_n^N$  exhibits the following output symmetry. For a randomly selected instance of  $f$  from  $F_n^N$ , the input transition that causes an output transition from  $y$  to  $y'$  is indistinguishable if the inputs come from the same group equivalence class. This prevents an adversary from extracting any information about differential input instances that caused an observed output change, which captures a typical attack scenario where only the output is observable. We will describe an attack method in Section 4.3.3.10 where through carefully controlled differential input pairs, it may be possible to infer almost all the top level gates. Output symmetry property prevents that occurrence. The preceding property makes it hard to distinguish between two vectors that cause a  $y \rightarrow y'$  change. Moreover, even a model that attempts to ascertain a specific input bit position transitions,  $x_i \rightarrow \bar{x}_i$  and  $x_j \rightarrow \bar{x}_j$ , is also indistinguishable since all the input bit positions cause the output  $y \rightarrow y'$  change uniformly. Quantitatively, let  $f \in_R F_n^N$ .

$$P[x_i \uparrow | (y, y')] = P[x_j \uparrow | (y, y')]$$

That is, the probability of switching for every input bit position is same for any pair of observed output when averaged over all the functions in the family  $F_n^N$ .

**Definition 4.3.9.** A function family  $F$  is output-symmetric if for a randomly chosen function instance  $f \in_R F$ , for a given output pair  $(y, y')$ ,  $P[x_i \uparrow | (y, y')] = P[x_j \uparrow | (y, y')]$  for all  $1 \leq i, j \leq N$ .

**Definition 4.3.10.** Two pairs of inputs  $(p, p')$  and  $(q, q')$  are in the same group-equivalence class iff the following condition holds.  $h_{\Delta_g}(p, p') = h_{\Delta_g}(q, q')$  which also implies  $\Delta_g(p, p') = \Delta_g(q, q')$ .

Let  $f \in_R F_n^N$ ,  $x, y \in I_N$  such that  $y = f(x)$ . Let  $x', x'' \in_R I_N$  such that the pairs  $(x, x')$  and  $(x, x'')$  are in the same group-equivalence class. Then given that  $y$  has changed to  $y'$  output-symmetry property of  $F_n^N$

requires that both these pairs be equally probable to have caused the transition. Let  $P^f[(y, y') | (x, x')]$  be the probability that  $y$  transitions to  $y'$  given that the input transitions from  $x$  to  $x'$ . Then from tree independence property and *input-symmetry* property (Properties 4.3.3.3 and 4.3.3.6),

$$P^f[(y, y') | (x, x')] = (P[\Gamma(x) = \Gamma(x') | (x, x')])^{N-h(y, y')} (1 - P[\Gamma(x) = \Gamma(x') | (x, x')])^{h(y, y')}$$

Let  $h'_i$  and  $h''_i$  be the hamming distance of the *group- $i$*  of the pairs  $(x, x')$  and  $(x, x'')$  respectively. Then the sequence  $h'_1, h'_2, \dots, h'_{N/n}$  is a permutation of the sequence  $h''_1, h''_2, \dots, h''_{N/n}$ . The collision probabilities of the level-1 (top level) gates are independent of their group numbers as the gates are chosen uniformly at random for every group position. Hence the collision probabilities of the tree gates for the pairs  $(x, x')$  and  $(x, x'')$  are equal. Thus,

$$P^f[(y, y') | (x, x')] = P^f[(y, y') | (x, x'')] \quad (4.6)$$

But,

$$P^f[(x, x') | (y, y')] = \frac{P^f[(y, y') | (x, x')] P[(x, x')]}{P^f[(y, y')]}$$

Using the result of Equation 4.6,

$$P^f[(x, x') | (y, y')] = \frac{P^f[(y, y') | (x, x'')] P[(x, x')]}{P^f[(y, y')]}$$

But,  $P[(x, x')] = P[(x, x'')]$  as both are in the same group-equivalence class. Thus,

$$P^f[(x, x') | (y, y')] = \frac{P^f[(y, y') | (x, x'')] P[(x, x'')]}{P^f[(y, y')]}$$

$$P^f[(x, x') | (y, y')] = P^f[(x, x'') | (y, y')] \quad (4.7)$$

The construction of group-equivalence class ensures that every group is equally probable and every bit within a group is equally probable or in other words the number of transitions for every bit within a group-equivalence class is same. Hence from Equation 4.7,

$$P^f[x_i \uparrow | (y, y')] = P^f[x_j \uparrow | (y, y')] \quad (4.8)$$

### 4.3.3.8 Neighborhood Probability:

**Definition 4.3.11 (Key hamming distance).** We define the hamming distance of key  $\kappa$  and  $\kappa'$  as the number of gate configurations they differ. The operator  $\Delta_k$  returns hamming distance of keys  $\kappa$  and  $\kappa'$ .

In order to use the function family  $F_n^N$  in cryptographic functions *neighborhood* property needs to be studied.

**Definition 4.3.12 ( $\{d_x, d_y, d_k\}$ -neighborhood probability).** Let  $x, y, \kappa$  and  $x', y', \kappa'$  be any two sets s.t  $y = f_\kappa(x)$  and  $y' = f_{\kappa'}(x')$ . Let  $h()$  be the bit-hamming distance function. Then their neighborhood probability is defined as  $p_{coll}^{\{d_x, d_y, d_k\}} = P[h(y, y') = d_y \mid \Delta_k(\kappa, \kappa') = d_k \cap h(x, x') = d_x]$ . Note that this probability is averaged over all the inputs and all possible key pairs.

Deriving a generic expression for  $\{d_x, d_y, d_k\}$ -neighborhood probability is very cumbersome. Hence we will derive only for  $d_k = 1$ . Consider the case when  $d_x = 0$  and  $d_y = 0$ . Consider a tree  $\Gamma$  in which only a level-1 gate is changed. Then,

$$\begin{aligned}
 P[\text{collision in } \Gamma] &= 1 - P[\text{no collision in } \Gamma] \\
 &= 1 - P[\text{no collision in top level} \cap \text{no collision in subsequent levels}] \\
 &= 1 - P[\text{no collision in top level}] \cdot P[\text{no collision in subsequent levels}] \\
 &= 1 - \left(\frac{1}{2} \cdot \left(1 - \frac{2^{n-1} - 1}{2^n - 1}\right) \log_n N - 1\right) \\
 p_{coll}^{\{0,0,1\}}(\Gamma) &= 1 - \left(\frac{1}{2} \cdot \left(\frac{2^{n-1}}{2^n - 1}\right) \log_n N - 1\right) \\
 &< 1 - \left(\frac{1}{2}\right)^{\log_n N}
 \end{aligned}$$

But every configuration in the key is reused  $\frac{N-1}{n-1}$  times. Hence,

$$\begin{aligned}
 p_{coll}^{\{0,0,1\}} &< P[\text{collision in } \Gamma]^{\frac{N-1}{n-1}} \\
 &< (p_{coll}^{\{0,0,1\}}(\Gamma))^{\frac{N-1}{n-1}} \\
 &< \left(1 - \left(\frac{1}{2}\right)^{\log_n N}\right)^{\frac{N-1}{n-1}}
 \end{aligned}$$

Note that in the above expression the inequality arises due to the following fact. Not all the  $\frac{N-1}{n-1}$  copies of the same configuration will be present in the top level and as the level where the gate is present increases

the collision probability decreases. Hence the collision probability calculated by keeping all the gates in the top level becomes upper bound.

Similarly,

$$p_{coll}^{\{0,d_y,1\}} < \binom{\frac{N-1}{n-1}}{d_y} \cdot (p_{coll}^{\{0,0,1\}}(\Gamma))^{\frac{N-1}{n-1}-d_y} \cdot \frac{1}{2^{d_y}}$$

$$p_{coll}^{\{0,d_y,1\}} < \binom{\frac{N-1}{n-1}}{d_y} \cdot \left(1 - \left(\frac{1}{2}\right)^{\log_n N}\right)^{\frac{N-1}{n-1}-d_y} \cdot \frac{1}{2^{d_y}} \quad (4.9)$$

The neighborhood probability for an input change of  $d_x$  and key hamming distance of 1 can be estimated as follows. From the Property 4.3.3.4 the maximum collision controllability is  $(1 - (\frac{2^{n-1}}{2^n-1})^{\log_n N})^N$ . Since the key has a hamming distance of 1,  $\frac{N-1}{n-1}$  configurations would have changed from  $f_k$  to  $f_{k'}$ . When the configurations are different the collision probability increase from  $\frac{2^{n-1}-1}{2^n-1}$  to  $\frac{1}{2}$ . Hence by taking the upper bound of the collision probabilities of the trees ( $\Gamma$ ) which are affected by the configuration change we can find upper bound to  $p_{coll}^{\{d_x,0,1\}}$ . Hence,

$$p_{coll}^{\{d_x,0,1\}} < \left(1 - \left(\frac{2^{n-1}}{2^n-1}\right)^{\log_n N}\right)^{N-\frac{N-1}{n-1}} \cdot \left(1 - \left(\frac{1}{2}\right)^{\log_n N}\right)^{\frac{N-1}{n-1}}$$

$$p_{coll}^{\{d_x,0,1\}} < \left(1 - \left(\frac{1}{2}\right)^{\log_n N}\right)^N \quad (4.10)$$

The other useful inequality is

$$p_{coll}^{\{d_x,d_y,1\}} \leq p_{coll}^{\{d_x,0,1\}} \quad (4.11)$$

This is because the collision probability of every tree is always greater than  $\frac{1}{2}$  and forms the upper bound.

#### 4.3.3.9 bias propagation:

From the Property 4.2.2.10 of the set  $G_n^b$  the *bias* at the output of a gate  $g \in G_n^b$  is less than the input bias. Thus the *bias* reduces as it propagates through the gates. In  $F_n^N$  the bias of every tree  $\Gamma$  is independent. In a tree there are  $\log_n N$  levels of gates hence the output bias is lesser than the input bias.

#### 4.3.3.10 Observability weakness:

One of the main design goals of  $F_n^N$  is to make every output bit as function of all the input bits. The second design goal is to minimize the number of configurations. These two design goals itself forms the weakness of the function family. Consider a black box implementation of  $f_{\kappa} \in_R F_n^N$ . In order to identify the secret (*configurations*) the approach would be to generate a pair of inputs  $x, x'$  such that they differ in only one group (say  $i$ ) *i.e.*,  $\Delta_g(x, x') = 1$ . Let  $y = f_{\kappa}(x), y' = f_{\kappa}(x')$  be the outputs and  $r = grp(x, i)$  and  $r' = grp(x', i)$ . Let  $b = y \oplus y'$  be the difference in the outputs and  $b_j$  represent  $j^{th}$  bit of  $b$ . Then if  $b_j = 1$  then the configuration in  $\gamma_{1,i}(\Gamma_i) = \kappa_k$  differs in rows  $r$  and  $r'$ , *i.e.*,  $\kappa_k[r] \neq \kappa_k[r']$ . The expected number of 1's in  $b$  is nothing but  $N \cdot (1 - p_{coll}^g)^{\log_n N}$ . Hence to find out about the characteristic of all the  $N$  configurations for the rows  $r$  and  $r'$  a minimum of  $\frac{1}{(1 - p_{coll}^g)^{\log_n N}}$  is needed. Hence by doing this experiment for all  $2^{n-1}$  unique pairs of  $r$  and  $r'$  it is possible to find out which rows are similar and which rows are different. Once the relationship between the rows are identified the key space reduces to  $2^{2N}$  bits. This is much smaller compared to the actual key space  $|G_n^b|^N$ .

#### 4.3.3.11 Strength of $f^{(2)}, f \in F_n^N$

**Observability:** The weakness described in Section 4.3.3.10 arises due to the fact that the inputs to the top level gates ( $\gamma_{1,i}(\Gamma)$ ) are controllable and the transitions (or switching) at the output of top level gates ( $y(\gamma_{1,i}(\Gamma))$ ) are observable at the output of the tree ( $y(\Gamma)$ ). Thus the controllability and observability should be *de-linked* in order to improve the strength of the function. Precisely this is achieved by the function composition  $f^{(2)}$ .

In  $f^{(2)}$ , let  $x$  be the input,  $y$  be the output, and  $z$  be the output at midpoint *i.e.*,  $y = f^{(2)}(x)$  and  $z = f(x)$ . The observable output  $y$  does not provide any information about the switching of any particular intermediate bit  $z_i$  since all of them are equally probable to switch as shown in *output-symmetry* property (Property 4.3.3.7). The function family  $F_n^N$  also is *input-symmetric* (Property 4.3.3.6) hence the adversary can not control the intermediate bits in a *biased* manner to observe its effect at the output. Thus the method used to extract information about top level gates truth table rows as explained in Section 4.3.3.10 is not applicable to  $f^{(2)}$ . We believe that given polynomial input output pairs of  $f^{(2)}$  deciphering  $f$  is an *NP-hard* problem. But we do not have a proof for this property at this time.

**Neighborhood probability dispersion:** The composition  $f^{(2)}$  or in general  $f^{(n)}$  decreases (or disperses) the neighborhood collision probability. In effect the composition makes the distribution more uniform and random. Let  $y = f_{\kappa}^{(2)}(x) = f_{\kappa}(z)$  where  $z = f_{\kappa}(x)$ . Then

$$\begin{aligned} p_{coll}^{\{0,d_y,1\}}(f^{(2)}) &= \sum_{i=0}^{\frac{N-1}{n-1}} P[h(y,y') = d_y | h(z,z') = i] \cdot P[h(z,z') = i] \\ &= \sum_{i=0}^{\frac{N-1}{n-1}} p_{coll}^{\{0,i,1\}} p_{coll}^{\{i,d_y,1\}} \end{aligned}$$

From Equations 4.9, 4.10, and 4.11,

$$\begin{aligned} \sum_{i=0}^{\frac{N-1}{n-1}} p_{coll}^{\{0,i,1\}} p_{coll}^{\{i,d_y,1\}} &\leq p_{coll}^{\{0,d_y,1\}} \sum_{i=0}^{\frac{N-1}{n-1}} p_{coll}^{\{0,i,1\}} \\ &\leq p_{coll}^{\{0,d_y,1\}} \end{aligned}$$

Thus,

$$p_{coll}^{\{0,d_y,1\}}(f^{(2)}) \leq p_{coll}^{\{0,d_y,1\}} \quad (4.12)$$

The other way to look at this is using the *bias* propagation property (Property 4.3.3.9). As more and more levels of gates are added the bias will be decreasing and the collision probability approaches  $\frac{1}{2}$  for every output bit.

#### 4.4 REBEL function family $R_n^{2N}$

**Definition 4.4.1** ( $R_n^{2N}$  function family).  $R_n^N$  is a  $I_{2N} \mapsto I_{2N}$  LR-network [37, 23] of  $f^{(2)}$  with 4 levels, where  $f \in F_n^N$ .

##### 4.4.1 Construction of $R_n^{2N}$

Let  $\kappa \in \mathcal{K}$  be the key that chooses the function  $f_{\kappa} \in F_n^N$ . Let  $g_{\kappa}$  be the function defined as follows,

$$g_{\kappa}(x_{|L} \bullet x_{|R}) = x_{|R} \bullet (x_{|L} \oplus f_{\kappa}^{(2)}(x_{|R}))$$

Let  $s$  be a swapping function, such that,  $s(x_{|L} \bullet x_{|R}) = x_{|R} \bullet x_{|L}$ .

Then the encryption function  $E_{\kappa} \in R_n^{2N}$  is constructed as,  $E_{\kappa} = g_{\kappa}^{(4)} \circ s$ . The decryption function  $D_{\kappa} \in R_n^{2N}$  is same as the  $E_{\kappa}$ . Figure 4.6 shows the diagrammatic representation of  $E_{\kappa}$  and  $D_{\kappa}$ .

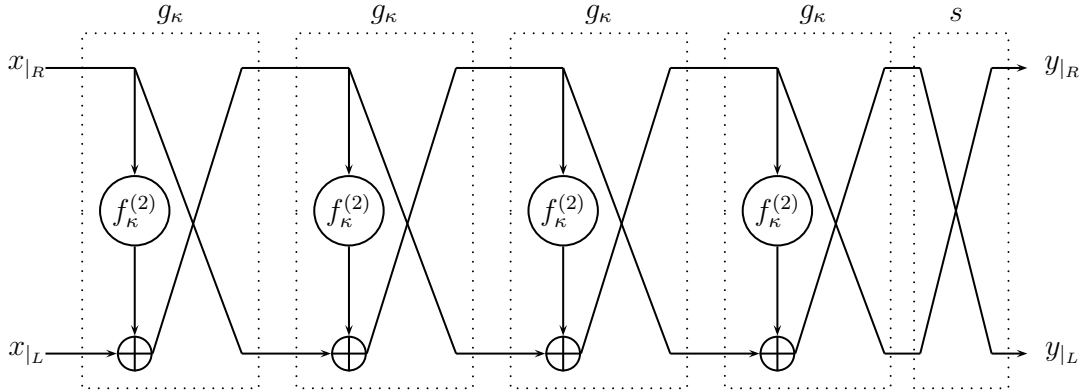


Figure 4.6 Diagrammatic Representation of  $E_{\kappa}$  and  $D_{\kappa}$  in  $R_n^{2N}$

#### 4.4.2 Adversary Models

The aim of the adversary in any cryptosystem is obtain an algorithm to extract the secret (or key) of the family. We define two kinds of adversary models distinguished based on its capabilities.

**Definition 4.4.2 (function oracle  $O_f$ ).** Oracle  $O_f$  performs encryption ( $E_{\kappa}$ ) and decryption ( $D_{\kappa}$ ) for a randomly chosen key  $\kappa \in \mathcal{K}$ . Every query to  $O_f$  is a tuple  $(x, e)$  where,  $x \in I_{2N}$  and  $e \in \{1, 0\}$ . The oracle returns  $y = e \cdot E_{\kappa}(x) + \bar{e} \cdot D_{\kappa}(x)$  where  $y \in I_{2N}$ .

**Definition 4.4.3 (unbounded statistics oracle  $O_s$ ).** Oracle  $O_s$  has access to the input-output table for the entire key-space  $\mathcal{K}$ . Specifically, the model maintained by  $O_s$ ,  $M(O_s)$  consists of a set of triples  $\{(x, y, \kappa) | x \xrightarrow{\kappa} y\}$  where  $x \xrightarrow{\kappa} y$  implies that  $x$  maps to  $y$  with key  $\kappa$ . The query to  $O_s$  is an arbitrary set  $Q$  of input-output pairs, potentially chosen adaptively, i.e.,  $Q = \{(x, y) | x, y \in I_{2N}\}$  such that  $|Q| = l_q$ . The oracle returns a set of keys  $\kappa_Q$  which are consistent with the input-output relations in the query set  $Q$ . Let  $|\kappa_Q| = l_r$ .

**Definition 4.4.4 (poly time statistics oracle  $O_s^P$ ).** Oracle  $O_s^P$  is similar to the unbounded statistics oracle  $O_s$  except that it only has polynomial time to build its static model of the input-output table with the key-space  $\mathcal{K}$  correlation. For instance, it can choose to develop the correlation for a given key  $\kappa_{i_0}$  with respect to polynomially many input, output pairs. Or it can choose to develop this correlation for a

constant number of input-output pairs for polynomially many key values. Hence the cardinality of its model,  $|M(O_s^P)|$ , is polynomially bounded. The query to  $O_s^P$  is a set of input-output pairs,  $Q = \{(x,y)|x,y \in I_{2N}\}$  such that  $|Q| = l_q$ . The number of queried input-output pairs  $l_q$  is of course polynomially bounded due to the poly time constraint. The oracle returns a set of keys  $\kappa_Q$  from within its model which are consistent with the input-output relations in the query set  $Q$ . If none match then an empty set is returned. Let  $|\kappa_Q| = l_r$ .

**Definition 4.4.5 (poly-time static adversary).** A polynomial time static adversary (PTSA) has access to both oracles  $O_f$  and  $O_s$ . The adversary performs  $p(N)$  queries to  $O_f$  for a polynomial  $p(N)$ . These queries help the adversary build a query set  $Q$  for  $O_s$ . The adversary has to decipher the secret (or key) based on the result of  $O_s$ .

**Definition 4.4.6 (poly-time runtime adversary).** A polynomial time runtime adversary (PTRA) has access only to  $O_f$ . The adversary performs polynomially in  $N$  many queries to  $O_f$  and based on the input-output relation has to infer the secret (or the key).

#### 4.4.3 Complexity Analysis (PTRA)

Let  $x, y \in I_{2N}$  and  $y = E_\kappa(x)$ . From the construction of  $E_\kappa$ ,

$$\begin{aligned} y_{|R} &= x_{|L} \oplus f_\kappa^{(2)}(x_{|R}) \oplus f_\kappa^{(2)}(x_{|R} \oplus f_\kappa^{(2)}(x_{|L} \oplus f_\kappa^{(2)}(x_{|R}))) \\ y_{|L} &= x_{|R} \oplus f_\kappa^{(2)}(x_{|L} \oplus f_\kappa^{(2)}(x_{|R})) \oplus f_\kappa^{(2)}(y_{|R}) \end{aligned}$$

In order to derive any information about the *key*, which is essentially configurations to the reconfigurable gates, the adversary has to be able to control the input and observe its effect on the output for these component gates. The construction of  $E_\kappa$  ensures that if the input to a gate is controllable then the output is not observable. Hence, the adversary with polynomially many trials can not learn any information about the input output relation of component gates.

The only observable effect of  $f_\kappa$  is its collision. If the adversary chooses a pair  $x_{|L}, x'_{|L}$  such that  $f_\kappa^{(2)}(x_{|R} \oplus f_\kappa^{(2)}(x_{|L} \oplus f_\kappa^{(2)}(x_{|R}))) = f_\kappa^{(2)}(x_{|R} \oplus f_\kappa^{(2)}(x'_{|L} \oplus f_\kappa^{(2)}(x_{|R})))$  then  $y_{|R} \oplus x_{|L} = y'_{|R} \oplus x'_{|L}$ . From the Property 4.3.3.4 of  $F_n^N$  the maximum collision probability occurs when the minimum number of gates on the top level are exercised. Let,  $x, x'$  be the input to the function  $f_\kappa$ , then the collision probability is maximum if the difference (or change) occurs in only one of the group ( $\Delta_g(x, x') = 1$ ).

The adversary can determine that two rows of all the  $N$  top level gates collide if such an event can be identified. But the existence of  $f_{\kappa}^{(2)}(x_{|R})$  masks input to the second function  $f_{\kappa}^{(2)}(x'_{|L} \oplus f_{\kappa}^{(2)}(x_{|R}))$ . Let  $E$  be the event such that the adversary can determine that all the  $N$  gates collide in two rows  $r, r'$ . This can be considered as partial key extraction, as this information could be used to further reduce the search space in a brute force attack. Let  $x_{|L}$  and  $x'_{|L}$  differ in *group*  $i$  and let  $grp_i$  operator return the  $i^{th}$  group. Then,  $P[E] = P[(x_{|L}, x'_{|L} \text{ cause collision in } N \text{ gates}) \cap (r = grp_i(x_{|L} \oplus f_{\kappa}(x_{|R})))]$ . Probability of the event  $r = grp_i(x_{|L} \oplus f_{\kappa}^{(2)}(x_{|R}))$  is nothing but  $\frac{1}{2^n}$ .

In order to find  $E$  the following experiment is performed. Choose  $x = x_{|L} \bullet x_{|R}$  uniformly at random from the set  $I_{2N}$ . Choose  $i \in [1, \frac{N}{n}]$  at random. Then generate  $\binom{2^n}{2}$  pairs of queries to  $O_f$  such that  $x'_{|L}$  differ with  $x_{|L}$  only in bin  $i$ . If a collision occurs at the output, the collision could be caused by the first level gates or the subsequent gates. To verify whether top level has caused the collision generate  $N$  pairs at random such that bin  $i$  has the same values as  $x_{|L}$  and  $x'_{|L}$ . If the collision is caused by top level gates then every pair will collide. If any one of these pairs does not cause collision then the top level gates have not collided.

Then,

$$P[E] = \binom{2^n}{2} \cdot \left(\frac{2^{n-1}-1}{2^n-1}\right)^N \cdot \frac{1}{2^n} < \frac{1}{2^{N-n+1}}$$

Note that this probability is not affected by the number of trials the adversary is performing, as it is determined by the properties of  $G_n^b$ .

Once the adversary has identified a collision between rows  $r$  and  $r'$ , the adversary has to guess the actual values of truth table configuration. And the probability of success is  $\frac{1}{2^n}$ . Hence, the probability of successfully guessing two rows of truth tables in all the  $N$  gates is bounded by  $\frac{1}{2^{2N-n+1}}$ .

$$P[\text{identifying two rows } r \text{ and } r' \text{ of } N \text{ gates}] < \frac{1}{2^{2N-n+1}}$$

#### 4.4.4 Statistical Adversary Advantage

In this section, we develop an attack framework for an adversary deploying the statistical oracles  $O_s$  and  $O_s^P$ .

A few observations are in order. For a given input-output pair  $(x, y)$ , the probability that the instantiated key  $\kappa_{inst}$  collides with another key  $\kappa$ , *i.e.*,  $P[E_{\kappa_{inst}}(x) = E_{\kappa}(x)] = \frac{1}{2^{2N}}$ . This follows from the Property 4.3.3.2.

Thus the expected number of keys that will collide for any given input-output pairs is  $\frac{|G_n^b|^N}{2^{2N}}$  and for  $k$  pairs,  $\frac{|G_n^b|^N}{2^{2Nk}}$ . Thus with polynomially many input-output pairs (for  $n = 4$ ,  $k \approx 6$ ) the adversary can get the correct key if the adversary has access to the unbounded statistics oracle  $O_s$ . However, the unbounded model  $M(O_s)$  will take unrealistic space and time resources. Hence, we will use the poly bounded oracle  $O_s^P$  instead.

Once the oracle is limited to build a model consisting of polynomially many tuples  $(x, y, \kappa)$ , it is very unlikely that it can provide the key association for a randomly chosen  $(x, y)$  by the adversary. Note that the adversary can evaluate an arbitrary instance of  $E_{\kappa_{inst}}(x)$  or  $D_{\kappa_{inst}}(y)$  through  $O_f$ . Hence we do not make any distinction between chosen plaintext or chosen ciphertext attacks. The adversary has to rely upon  $O_s^P$  to extract any key association with the input-output pairs. The  $(x, y, \kappa)$  space is very large,  $2^{2N} * 2^{2N} * |G_n^b|^N$ . Hence, a polynomially bounded oracle has negligible probability that any triples held by it that match on  $(x, y)$  pair will also match with respect to the key  $\kappa_{inst}$ . This leads to  $O_s^P$ 's inability to help the adversary. The most interesting issue here, then, is what other information can  $O_s^P$  provide the adversary, even though it does not contain a consistent  $(x, y, \kappa)$  triple. Any such information is useful if it infers a significant fraction of the key bits.

The *neighborhood* property of the  $E_\kappa$  can be exploited by the model in this respect. The reason such neighborhoods help the adversary is that the oracle does not need to have an exact match for  $(x, y)$  in its model any more (which is an unlikely event as we argued). Even if it contains a triple  $(x + \Delta x, y + \Delta y, \kappa)$  whose input-output component is from the neighborhood of the query  $(x, y)$ , it can report back the key of this triple,  $\kappa$ , to the adversary. The adversary gains significant information about  $\kappa_{inst}$  from  $\kappa$  since it is likely that  $\kappa_{inst} = \kappa + \Delta \kappa$ . Depending on the size of the neighborhood,  $\Delta \kappa$ , the adversary has a much more tractable problem at hand. The efficiency of this exploration depends on the structure of these neighborhoods. Figure 4.7 illustrates this issue.

Smaller the neighborhoods, more unlikely it is for the oracle to hold a matching triple in its model from such a neighborhood of the query. Larger neighborhoods will make it more likely that a matching triple from such a neighborhood will be found by the oracle. The adversary will, however, then have to explore the entire neighborhood bubble to determine the full key  $\kappa_{inst}$ . What we will show in this section is that such neighborhoods do not exist in the proposed function family with large probabilities. This makes

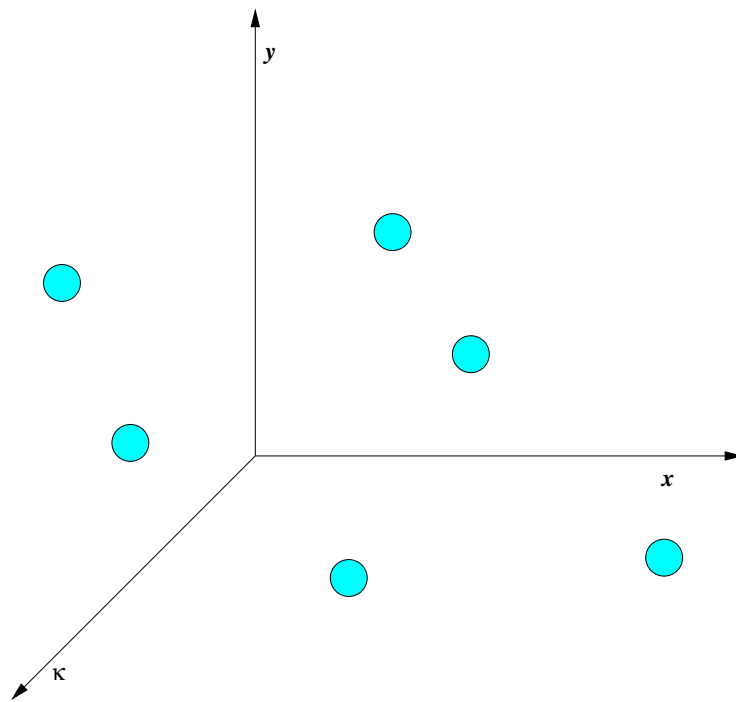


Figure 4.7  $(x, y, \kappa)$  Neighborhood Bubbles

any experimentation with  $f_{\kappa}$  equivalent to a random experiment. In other words, success probabilities for an adversary even with a poly bounded statistical oracle are almost as good as they would have been with a random search for the key.

From Properties 4.3.3.8 and 4.3.3.11 we see that the neighborhood probability gets dispersed with more levels of function compositions. In  $E_{\kappa}$  at least 4 levels of function ( $f_{\kappa}$ ) compositions are there in both the paths (left and right). Hence the neighborhood probabilities estimated for  $f_{\kappa}$  will form the upper bound for  $E_{\kappa}$ .

**Definition 4.4.7 (modified poly bounded oracle).** *We modify the poly bounded  $O_s^P$  oracle as follows. On a query  $(x, y)$ , it returns a triple within its model  $(x_{(i)}, y_{(i)}, \kappa_{(i)}) \in M(O_s^P)$  such that  $h(x, x_{(i)}) + h(y, y_{(i)})$  is minimized over  $\forall i$ .*

**Lemma 4.4.1 (statistical adversary advantage).** *A poly bounded adversary with a poly bounded oracle has probability at most  $\frac{N^c}{(1 - (\frac{1}{2})^{\log_n N})^N \cdot 2^{2N} \cdot 2^{2N} \cdot |G_n^b|^N}$ .*

The proof follows from Equation 4.10 and from the fact that  $(x, y, \kappa)$  space has size  $2^{2N} * 2^{2N} * |G_n^b|^N$ . The main point to note here is that the adversary is almost as well off as it would be in a completely random experiment. Hence, such a polynomially resource bounded oracle adds marginally to an adversary's capability for the REBEL functions due to lack of any non-uniformity.

#### 4.4.5 Resilience to Cryptanalysis

In this section we investigate the resilience of *REBEL* functions towards the well known cryptanalysis methods. First we will show the class of attacks which use the *invariance* property of system, *i.e.*, the idea of these attacks is to find the properties of the system which are not dependent or least dependent either on the secret or the input.

##### 4.4.5.1 Linear Cryptanalysis

*Linear cryptanalysis* is a general form of cryptanalysis based on finding affine approximations to the cipher function. The technique [38] has been applied to attack ciphers like FEAL [39] and DES [41]. Linear cryptanalysis exploits the high probability of occurrences of linear expressions involving *plaintext*, *ciphertext*, and *sub-key* bits. This attack becomes possible on the *conventional* cipher function design

as the *key* bits are primarily XOR'ed with round inputs. And linear approximations of known components (S-boxes) in the system further aid the analysis. In the case of *REBEL* none of these required conditions are available. Every gate (*component gates of tree*) is chosen by *key* and hence no linear approximation is possible.

#### 4.4.5.2 Differential Cryptanalysis

*Differential cryptanalysis* [6] exploits the property of difference being propagated from input to the output of the cipher function. This attack again requires the properties of the known components in the system (S-boxes) in order to estimate the difference propagation probabilities. In *REBEL* such an analysis is not possible as there are no known components in the system. A variant to this attack is *impossible difference attack* [7] which again uses the principle of identifying differences that does not propagate from input to output.

#### 4.4.5.3 Boomerang Attack

This attack [55] relies on the difference analysis of round function properties and existence of some block in the system which is independent of the input to cipher function. This can be thought of as meet-in-the-middle version of differential cryptanalysis. Again *REBEL* is resistant as there are no blocks (gates) in the system that is either independent of key or the input.

#### 4.4.5.4 Sliding Attack

This attack [8] exploits the weakness of the round functions. It assumes that given two pairs  $P, C$  and  $P', C'$  such that  $P' = f(P)$  and  $C' = f(C)$  then the round function can be deciphered or at least significant bits of the keys can be extracted. These attacks once again uses the property of round functions being built using some know components (S-boxes) and *key* bits being used only in XOR operations.

In *REBEL* the round function constitute  $f_k^{(2)}$ . As shown in Section 4.3.3.11 the round function can not be deciphered given a polynomial sized set of input output pairs. Hence sliding attack is also ineffective against *REBEL*.

#### 4.4.6 Implementation:

The implementation of *REBEL* requires the usage of  $n$ -to-1 reconfigurable gates. These reconfigurable gates are implemented as  $2^n$ -to-1 *multiplexers* and hence the size (as well as delay) increases exponentially with the increase in  $n$ . The practicable value of  $n$  is 4, as less than that will not have sufficiently rich gate space, and more than that will have very high area and delay penalty. Table 4.1 lists the properties of two instances of the *REBEL* function family.

Table 4.1 Properties of two instances of  $R_n^{2N}$

Property	$R_4^{32}$	$R_4^{128}$
$ G_4^b $	12870	12870
key size (bits)	256	1024
$ \mathcal{K} $	$> 2^{218}$	$> 2^{873}$
$\delta_c$	$\left(1 - \left(\frac{8}{15}\right)^2\right)^{16}$	$\left(1 - \left(\frac{8}{15}\right)^3\right)^{64}$
$P_{coll}^f$	$\approx \frac{1.5}{2^{16}}$	$\approx \frac{1.075}{2^{64}}$

## 4.5 Conclusion

In this chapter we have provided a novel design approach towards constructing block cipher functions using reconfigurable gates. We also provided detailed analysis of various properties of *REBEL* function family.

## CHAPTER 5. CONCLUSION

Software protection is one of the most important problems in the area of computing. However, software only solutions (such as compiler transformations of control flow or insertion of redundant basic blocks or data structure transformations) often do not have robustness of crypto methods. Complete control flow obfuscation methods have the limitation that they cannot hide the correct control flow information from the prying eyes of the OS/end user.

We propose a minimal architecture, *Arc3D*, to support efficient obfuscation of both static binary file system image and dynamic execution traces. This obfuscation covers three aspects: address sequences, contents, and second-order address sequences (patterns in address sequences exercised by the first level of loops). A robust obfuscation also prevents tampering by rejecting a tampered instruction at an adversary desired program point with an extremely high probability. Hence obfuscation and derivative tamper-resistance provide IP-protection. Consequently, *Arc3D* offers complete architecture support for copy-protection and IP-protection.

In TIVA we propose a novel solution to tackle the problem of verification. First we identify the relationship of IP protection with verification and provide a solution to solve both the problems in Embedded devices. TIVA verification mechanism satisfies all the requirements of a verification system without compromising the IP of the system being verified. We demonstrate that the silicon area overhead for TIVA is minimal, 1%, and its time overhead is completely absorbed in the pipeline.

We propose a novel design paradigm of cryptographic functions using reconfigurable gates and prove its cryptanalytic complexity. The advantages of such a design are manifold. The hardware implementation of REBEL is much faster when compared to the existing cryptographic functions, and it allows the use of a much larger secret (key) size than the block size. Since the future computing platforms essentially will contain some reconfigurable elements this design paradigm could be potentially beneficial. Even more

importantly, REBEL seems to be more robust against cryptanalysis than the traditional block ciphers, since it uses the secrets more effectively.

## REFERENCES

- [1] AOL. The America Online Instant Messenger Application. URL: <http://www.aol.com>  
Last Checked: 19 April 2006.
- [2] Arvind Seshadri and Adrian Perrig and Leendert van Doorn and Pradeep K. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *IEEE Symposium on Security and Privacy*, pages 272–, 2004. URL: <http://doi.ieeecomputersociety.org/10.1109/SECPRI.2004.1301329> Last Checked: 19 April 2006.
- [3] David Aucsmith. Tamper Resistant Software: An Implementation. In *Information Hiding*, pages 317–333, 1996.
- [4] David Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, pages 317–333, 1996.
- [5] R. Bennett and R. Landauer. Fundamental Physical Limits of Computation. *Scientific American*, pages 48–58, July 1985.
- [6] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *J. Cryptology*, 4(1):3–72, 1991.
- [7] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. *J. Cryptology*, 18(4):291–311, 2005.
- [8] Alex Biryukov and David Wagner. Slide attacks. In *Fast Software Encryption*, pages 245–259, 1999.
- [9] BPTM. Berkeley Predictive Technology Model. URL: <http://www-device.eecs.berkeley.edu>  
Last Checked: 19 April 2006.
- [10] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Department of Computer Science, University of Wisconsin-Madison, 1997.
- [11] Business Software Alliance. 8th Annual BSA Global Software Piracy Study. Trends in Software Piracy 1994-2002, 2003. URL: <http://global.bsa.org/> Last Checked: 19 April 2006.
- [12] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. In *Proceedings of International Security Conference (ISC)*, pages 144–155. Lecture Notes in Computer Science, 2200, Springer-Verlag, 2001.
- [13] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an aes implementation. In *Selected Areas in Cryptography*, pages 250–270, 2002. URL: <http://www.springerlink.com/link.asp?id=umthlwnv19tdqw8v> Last Checked: 19 April 2006.

- [14] Christian Collberg and Clark Thomborson and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997. URL: <http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html> Last Checked: 19 April 2006.
- [15] F. Cohen. Operating System Protection Through Program Evolution. *Computers and Security*, 12(6): 565–584, October 1993.
- [16] Christian S. Collberg and Clark D. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection. *IEEE Trans. Software Eng.*, 28(8):735–746, 2002. URL: <http://dx.doi.org/10.1109/TSE.2002.1027797> Last Checked: 19 April 2006.
- [17] Christian S. Collberg and Clark D. Thomborson. Software watermarking: Models and dynamic embeddings. In *POPL*, pages 311–324, 1999. URL: <http://doi.acm.org/10.1145/292540.292569> Last Checked: 19 April 2006.
- [18] ComputerWeekly.com. U.S. Software Security Takes Off, November 2002. URL: <http://www.computerweekly.com/Article117316.htm> Last Checked: 19 April 2006.
- [19] Zarka Cvetanovic and Richard E. Kessler. Performance analysis of the Alpha 21264-based Compaq ES40 system. In *ISCA*, pages 192–202, 2000. URL: <http://doi.acm.org/10.1145/339647.339680> Last Checked: 19 April 2006.
- [20] Dallas Semiconductor. DS5002 Secure Microprocessor Chip, March 2003. URL: <http://pdfserv.maxim-ic.com/en/ds/DS5002FP.pdf> Last Checked: 19 April 2006.
- [21] André DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, Los Alamitos, CA, 1994. IEEE Computer Society Press.
- [22] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34(10):57–66, 2001. URL: <http://doi.ieeecomputersociety.org/10.1109/2.955100> Last Checked: 19 April 2006.
- [23] H. Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, May 1973.
- [24] E. Fredkin and T. Toffoli. Conservative Logic. *International Journal of Theoretical Physics*, 21(3/4), April 1982.
- [25] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-612-9. URL: <http://doi.acm.org/10.1145/586110.586132> Last Checked: 19 April 2006.
- [26] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996. URL: <http://doi.acm.org/10.1145/233551.233553> Last Checked: 19 April 2006.
- [27] HMAC. HMAC: Internet RFC 2104, February 1997. URL: <http://www.rfc-archive.org/getrfc.php?rfc=2104> Last Checked: 19 April 2006.

- [28] HSPICE. Star-HSPICE 2001.4 Avant! Corporation.
- [29] IBM. IBM Power PC Data Sheets. URL: <http://www-306.ibm.com/chips/techlib/techlib.nsf/products/> Last Checked: 19 April 2006.
- [30] INTEL. Intel PCA Processors Data Sheets. URL: <http://www.intel.com/design/pca/applicationsprocessors/index.htm> Last Checked: 19 April 2006.
- [31] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003. URL: <http://springerlink.metapress.com/link.asp?id=lp3nma6cyx76cuec> Last Checked: 19 April 2006.
- [32] Markus Kuhn. The TrustNo1 Cryptoprocessor Concept. Technical Report 1997-04-30, Purdue University, April 1997.
- [33] Markus G. Kuhn. Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP. *IEEE Trans. Computers*, 47(10):1153–1157, 1998.
- [34] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS*, pages 168–177, 2000. URL: <http://doi.acm.org/10.1145/356989.357005> Last Checked: 19 April 2006.
- [35] Cullen Linn and Saumya K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Conference on Computer and Communications Security*, pages 290–299, 2003. URL: <http://doi.acm.org/10.1145/948109.948149> Last Checked: 19 April 2006.
- [36] Cullen Linn and Saumya K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Conference on Computer and Communications Security*, pages 290–299, 2003. URL: <http://doi.acm.org/10.1145/948149> Last Checked: 19 April 2006.
- [37] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, 1988.
- [38] Mitsuru Matsui. Linear Cryptanalysis Method for DES Cipher. In *EUROCRYPT*, pages 386–397, 1993.
- [39] Shoji Miyaguchi. The FEAL Cipher Family. In *CRYPTO*, pages 627–638, 1990.
- [40] National Bureau of Standards. FIPS PUB 197: Advanced Encryption Standard (AES). *Federal Information Processing Standard*, Nov 2001.
- [41] National Bureau of Standards. FIPS PUB 46-3: Data Encryption Standard (DES). *Federal Information Processing Standard*, May 1999.
- [42] NGSCB. Next-generation secure computing base, 2003. URL: <http://www.microsoft.com/ngscb> Last Checked: 19 April 2006.
- [43] T. Ogsio, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE Trans. on Fundamentals.*, E86(A)(1), January 2003.
- [44] PyxisSystemsTechnologies. AIM/oscar Protocol Specification: Section 3: Connection Management, 2002. URL: <http://www.oilcan.org/oscar/section3.html> Last Checked: 19 April 2006.

- [45] Rick Kennell and Leah H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *12<sup>th</sup> USENIX Security Symposium*, 2003.
- [46] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcb-based integrity measurement architecture. In *USENIX Security Symposium*, pages 223–238, 2004. URL: <http://www.usenix.org/publications/library/proceedings/sec04/tech/sailer.html> Last Checked: 19 April 2006.
- [47] Umesh Shankar, Monica Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *USENIX Security Symposium*, pages 89–102, 2004. URL: <http://www.usenix.org/publications/library/proceedings/sec04/tech/shankar.html> Last Checked: 19 April 2006.
- [48] Standard Performance Evaluation Corporation. Specbench: SPEC 2000 Benchmarks Version 1.3. URL: <http://www.specbench.org/osg/cpu2000/> Last Checked: 19 April 2006.
- [49] Steven J. E. Wilton and Norman P. Jouppi. An Enhanced Access and Cycle Time model for On-Chip Caches. In *WRL Research Technical Report 93/5*, July 1994.
- [50] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS*, pages 160–171, 2003. URL: <http://doi.acm.org/10.1145/782814.782838> Last Checked: 19 April 2006.
- [51] T. Toffoli. Reversible Computing. Technical Report MIT/LCS/TM151/1980, MIT Laboratory for Computer Science, 1980.
- [52] Trusted Computing Group. Trusted Platform Module, 2003. URL: <http://www.trustedcomputing.org> Last Checked: 19 April 2006.
- [53] Trusted Platform Module. TPM Design Principles - Version 1.2, October 2003. URL: <https://www.trustedcomputinggroup.org/specs/TPM> Last Checked: 19 April 2006.
- [54] TSMC. Taiwan Semiconductor Manufacturing Company Ltd. URL: <http://www.tsmc.com> Last Checked: 19 April 2006.
- [55] David Wagner. The boomerang attack. In *Fast Software Encryption*, pages 156–170, 1999.
- [56] Steve R. White and Liam Comerford. ABYSS: An Architecture for Software Protection. *IEEE Trans. Software Eng.*, 16(6):619–629, 1990. URL: <http://www.computer.org/tse/ts1990/e0619abs.htm> Last Checked: 19 April 2006.
- [57] XCELL Journal Online. Is Your FPGA Design Secure. URL: [http://www.xilinx.com/publications/xcellonline/xcell\\_47/xc\\_secure47.htm](http://www.xilinx.com/publications/xcellonline/xcell_47/xc_secure47.htm) Last Checked: 19 April 2006.
- [58] XSC. Intel 80200 Processor based on Intel XSCALE Microarchitecture Datasheet, January 2003.
- [59] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: An Infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS*, pages 72–84, 2004. URL: <http://doi.acm.org/10.1145/1024403> Last Checked: 19 April 2006.

## ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Akhilesh Tyagi for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work.

The people most responsible for my success at higher studies are my parents. They made me understand the importance of education, and guided me through the initial years of schooling. I also thank my parents and folks for their sustained support despite all the difficulties at home, that helped me see this day.

The technical discussions I had with my lab-mates, Sriram Nadathur, Pramod Rama Rao, Veerendra Allada, Ka-Ming Keung, and Swamy Ponpandi, have helped my research to a great extent. I also thank my lab-mates and Srivatsan Balasubramanian for their support during my thesis presentation preparation. I thank my friends, Harisudhakar Vepadharmalingam, Samarth Shetty, Girish Lingappa, Aravind Velayutham, Venkatesh Selvaraj, Satya Sarippalli, Ganesh T S, Anupreet Khaur, Anantharaman Kalyanaraman, Kirti Rajagopalan, Srivatsan Balasubramanian, Vishwanath Somashekar, Mahesh Narayanan, Poornima, Satyadev Nandakumar, Balaji Venkatachalam, Sai Sudhir, Nitin Jose, Lakshmi Narasimhan, Sruthi Sagar, Sudip Seal and many more, for making my stay at Ames a memorable one.